(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 847 552 B1

(12) **EUROPEAN PATENT SPECIFICATION**

(45) Date of publication and mention of the grant of the patent:
30.10.2002 Bulletin 2002/44

(21) Application number: **96930495.5**

(22) Date of filing: **07.08.1996**

(51) Int Cl.[7]: **G06F 7/00**, G06F 7/52, G06F 15/00, G06F 15/76, G06F 15/78, G06F 15/80, G06F 7/544

(86) International application number:
PCT/US96/12799

(87) International publication number:
WO 97/008610 (06.03.1997 Gazette 1997/11)

(54) **AN APPARATUS FOR PERFORMING MULTIPLY-ADD OPERATIONS ON PACKED DATA**

MULTIPLIXIER-ADDIERUNGSVORRICHTUNG FÜR GEPACKTE DATEN

DISPOSITIF DE MULTIPLICATION-ADDITION DE DONNEES COMPRIMEES

(84) Designated Contracting States:
CH DE ES FI FR GB IE IT LI NL SE

(30) Priority: **31.08.1995 US 522067**
23.02.1996 US 606212

(43) Date of publication of application:
17.06.1998 Bulletin 1998/25

(73) Proprietor: **INTEL CORPORATION**
Santa Clara, CA 95052 (US)

(72) Inventors:
- **PELEG, Alexander, D.**
  31015 Haifa (IL)
- **MITTAL, Millind**
  S. San Francisco, CA 94080 (US)
- **MENNEMEIER, Larry, M.**
  Boulder Creek, CA 95006 (US)
- **EITAN, Benny**
  31015 Haifa (IL)
- **DULONG, Carole**
  Saratoga, CA 95070 (US)
- **KOWASHI, Eiichi**
  Ryugasaki 301 (JP)
- **WITT, Wolf**
  Walnut Creek, CA 94598 (US)
- **LIN, Derrick, Chu**
  Foster City, CA 94404 (US)
- **BINDAL, Ahmet**
  Sunnyvale, CA 94086 (US)
- **FISHER, Stephen, A.**
  Boulder Creek, CA 95006 (US)
- **BUI, Tuan, H.**
  Phoenix, AZ 85048 (US)

(74) Representative: **Wombwell, Francis**
Potts, Kerr & Co.
15, Hamilton Square
Birkenhead Merseyside L41 6BR (GB)

(56) References cited:
EP-A- 0 308 124 **US-A- 4 771 379**
US-A- 5 442 799

- **JULIE SHIPNES, Graphics Processing with the 8810 RISC Microprocessor, IEEE 1992, pages 169-174.**

EP 0 847 552 B1

Description

BACKGROUND OF THE INVENTION

FIELD OF THE INVENTION

[0001] In particular, the invention relates to the field of computer systems. More specifically, the invention relates to the area of packed data operations.

DESCRIPTION OF RELATED ART

[0002] In typical computer systems, processors are implemented to operate on values represented by a large number of bits (e.g., 64) using instructions that produce one result. For example, the execution of an add instruction will add together a first 64-bit value and a second 64-bit value and store the result as a third 64-bit value. However, multimedia applications (e.g., applications targeted at computer supported co-operation (CSC - the integration of teleconferencing with mixed media data manipulation, 2D/3D graphics, image processing, video compression/decompression, recognition algorithms and audio manipulation) require the manipulation of large amounts of data which may be represented in a small number of bits. For example, graphical data typically requires 8 or 16 bits and sound data typically requires 8 or 16 bits. Each of these multimedia applications requires one or more algorithms, each requiring a number of operations. For example, an algorithm may require an add, compare and shift operation.

[0003] To improve efficiency of multimedia applications (as well as other applications that have the same characteristics), processors may provide packed data formats. A packed data format is one in which the bits typically used to represent a single value are broken into a number of fixed sized data elements, each of which represents a separate value. For example, a 64-bit register may be broken into two 32-bit elements, each of which represents a separate 32-bit value. In addition, these processors provide instructions for separately manipulating, in response to a single instruction, each element in these packed data types in parallel. For example, a packed add instruction adds together corresponding data elements from a first packed data and a second packed data. Thus, if a multimedia algorithm requires a loop containing five operations that must be performed on a large number of data elements. it is desirable to pack the data and perform these operations in parallel using packed data instructions. In this manner, these processors can more efficiently process multimedia applications.

[0004] However, if the loop of operations contains an operation that cannot be performed by the processor on packed data (i.e., the processor lacks the appropriate instruction), the data will have to be unpacked to perform the operation. For example, if the multimedia algorithm requires an add operation and the previously described packed add instruction is not available, the programmer must unpack both the first packed data and the second packed data (i.e., separate the elements comprising both the first packed data and the second packed data), add the separated elements together individually, and then pack the results into a packed result for further packed processing. The processing time required to perform such packing and unpacking often negates the performance advantage for which packed data formats are provided. Therefore, it is desirable to incorporate in a computer system a set of packed data instructions that provide all the required operations for typical multimedia algorithms. However, due to the limited die area on today's general purpose microprocessors, the number of instructions which may be added is limited. Therefore, it is desirable to invent instructions that provide both versatility (i.e. instructions which may be used in a wide variety of multimedia algorithms) and the greatest performance advantage.

[0005] One technique for providing operations for use in multimedia algorithms is to couple a separate digital signaling processor (DSP) to an existing general purpose processor (e.g., The Intel® 486 manufactured by Intel Corporation of Santa Clara, CA). The general purpose processor allocates jobs that can be performed using packed data (e.g., video processing) to the DSP.

[0006] One such DSP includes a multiply accumulate instruction that adds to an accumulation value the results of multiplying together two values. (see Kawakami, Yuichi, et al., "A Single-Chip Digital Signal Processor for Voiceband Applications", IEEE International Solid-State Circuits Conference, 1980, pp. 40-41). An example of the multiply accumulate operation for this DSP is shown below in Table 1, where the instruction is performed on the data values A_1 and B_1 accessed as Source1 and Source2, respectively.

Table 1

Multiply-Accumulate Source1. Source2

A_1	Source1
B_1	Source2
=	
A_1B_1+Accumulation Value	Result1

[0007] One limitation of this instruction is its limited efficiency -- i.e., it only operates on 2 values and an accumulation value. For example, to multiply and accumulate two sets of 2 values requires the following 2 instructions performed serially: 1) multiply accumulate the first value from the first set, the first value from the second set, and an accumulation value of zero to generate an intermediate accumulation value; 2) multiply accumulate the second value from the first set, the second value from the second set, and the intermediate accumulation value to generate the result.

[0008] Another DSP includes a multiply accumulate instruction that operates on two sets of two values and an accumulation value (See "Digital Signal Processor with Parallel Multipliers", patent number 4,771,470 - referred to herein as the "Ando et al." reference). An example of the multiply accumulate instruction for this DSP is shown below in Table 2, where the instruction is performed on the data values A_1, A_2, B_1 and B_2 accessed as Source1-4, respectively.

Table 2

Source1	Source3
A_1	A_2
Multiply Accumluate	
Source2	Source4
B_1	B_2
=	Result1
$A_1 \cdot B_1 + A_2 \cdot B_2$ + Accumulation Value	

Using this technique, two sets of 2 values are multiplied and then added to an accumulation value in one instruction.

[0009] This multiply accumulate instruction has limited versatility because it always adds to the accumulation value. As a result, it is difficult to use the instruction for operations other than multiply accumulate. For example, the multiplication of complex numbers is commonly used in multimedia applications. The multiplication of two complex number (e.g., r_1 i_1 and r_2 i_2) is performed according to the following equation:

$$\text{Real Component} = r_1 \cdot r_2 - i_1 \cdot i_2$$

$$\text{Imaginary Component} = r_1 \cdot i_2 + r_2 \cdot i_1$$

This DSP cannot perform the function of multiplying together two complex numbers using one multiply accumulate instruction.

[0010] The limitations of this multiply accumulate instruction can be more clearly seen when the result of such a

calculation is needed in a subsequent multiplication operation rather than an accumulation. For example, if the real component were calculated using the DSP, the accumulation value would need to be initialized to zero in order to correctly compute the result. Then the accumulation value would again need to be initialized to zero in order to calculate the imaginary component. To perform another complex multiplication on the resulting complex number and a third complex number (e.g., r3, i3), the resulting complex number must be rescaled and stored into the acceptable memory format and the accumulation value must again be initialized to zero.

[0011] Then, the complex multiplication can be performed as described above. In each of these operations the ALU, which is devoted to the accumulation value, is superfluous hardware and extra instructions are needed to re-initialise this accumulation value. These extra instructions would otherwise have been unnecessary.

[0012] A further limitation of this technique is that the data must be accessed through expensive multi-ported memory. This is because the multipliers are connected directly with data memories. Therefore the amount of parallelism which can be exploited is limited to a small number by the cost of the interconnection, and the fact that this interconnection is not decoupled from the instruction.

[0013] The Ando, et al, reference also describes that an alternative to this expensive interconnection is to introduce a delay for each subsequent pair of data to be multiplied. This solution diminishes any performance advantages to those provided by the solution previously shown in Table 1,

[0014] Furthermore, the notion of multi-ported memory or of pipelined accesses to memory entails the use of multiple addresses. This explicit use of one address per datum, clearly demonstrates that the critical notion of packed data is not employed in this technique.

[0015] Further, in EP-A-0308124 (Visual Information Technologies, Inc.), a high-speed image processing computer is described.

SUMMARY OF THE INVENTION

[0016] According to the invention, a processor having a first and second storage having a first and second packed data, respectively is described and claimed in the appended claims. Each packed data includes a first, second, third, and fourth data element.

[0017] A multiply-add circuit is coupled to the first and second storage areas. The multiply-add circuit includes a first, second, third and fourth multiplier, wherein each of the multipliers receives a corresponding set of said data elements. The multiply-add circuit further includes a first adder coupled to the first and second multipliers, and second adder coupled to the third and fourth multipliers. A third storage area is coupled to the adders. The third storage area includes a first and second field for saving output of the first and second adders, respectively, as first and second data elements of a third packed data.

BRIEF DESCRIPTION OF THE DRAWINGS

[0018]

Figure 1 illustrates an computer system having one embodiment of the invention.

Figure 2 illustrates a register file of the processor according to one embodiment of the invention.

Figure 3 is a flow diagram illustrating the general steps used by the processor to manipulate data according to one embodiment of the invention.

Figure 4 illustrates packed data-types according to one embodiment of the invention.

Figure 5a illustrates in-register packed data representations according to one embodiment of the invention.

Figure 5b illustrates in-register packed data representations according to one embodiment of the invention.

Figure 5c illustrates in-register packed data representations according to one embodiment of the invention.

Figure 6a illustrates a control signal format for indicating the use of packed data according to one embodiment of the invention.

Figure 6b illustrates a second control signal format for indicating the use of packed data according to one embodiment of the invention.

Figure 7 is a flow diagram illustrating steps for performing a multiply-add operation on packed data according to one embodiment of the invention.

Figure 8 illustrates a circuit for performing multiply-add operations on packed data according to one embodiment of the invention.

Figures 9a - 9e illustrate a Wallace Tree performing the partial product summation and reduction for one embodiment of the present invention.

Figures 10a - 10af illustrate one embodiment of a circuit implementing the Wallace Tree of figures 9a - 9e for one embodiment of the present invention.

Figure 11 illustrates a circuit for performing multiply-add operations on packed data according to one embodiment of the invention.

DETAILED DESCRIPTION

[0019] In the following description, numerous specific details are set forth to provide a thorough understanding of the invention. However, it is understood that the invention may be practiced without these specific details. In other instances, well-known circuits, structures and techniques have not been shown in detail in order not to obscure the invention.

DEFINITIONS

[0020] To provide a foundation for understanding the description of the embodiments of the invention, the following definitions are provided.

Bit X through Bit Y:	defines a subfield of binary number. For example, bit six through bit zero of the byte 00111010_2 (shown in base two) represent the subfield 111010_2. The '2' following a binary number indicates base 2. Therefore, 10002 equals 8_{10}, while F_{16} equals 15_{10}.
R_x:	is a register. A register is any device capable of storing and providing data. Further functionality of a register is described below. A register is not necessarily, included on the same die or in the same package as the processor..
SRC1, SRC2, and DEST:	identify storage areas (e.g., memory addresses, registers, etc.)
Source1-i and Result1-i:	represent data.

OVERVIEW

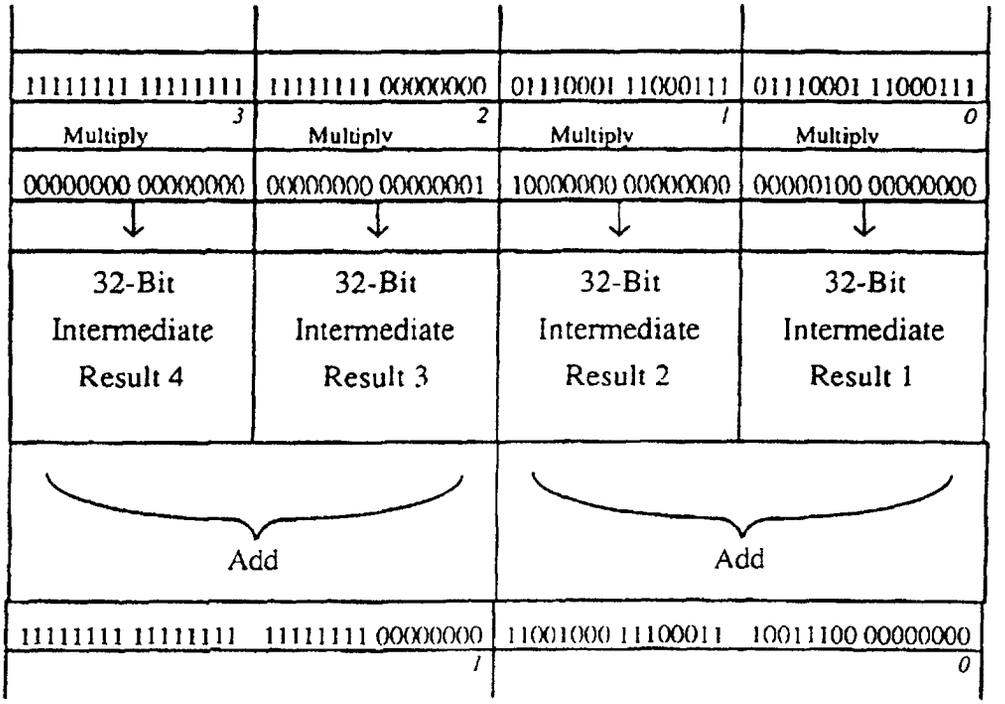
[0021] This application describes an apparatus in a processor performing multiply-add operations on packed data. In one embodiment, two multiply-add operations are performed using a single multiply-add instruction as shown below in Table 3a and Table 3b -- Table 3a shows a simplified representation of the disclosed multiply-add operation, while Table 3b shows a bit level example of the disclosed multiply-add operation.

Table 3a

Multiply-Add Source1, Source2

A_1	A_2	A_3	A_4	Source1
B_1	B_2	B_3	B_4	Source2
		=		
$A_1B_1+A_2B_2$		$A_3B_3+A_4B_4$		Result1

Table 3b



[0022] Thus, the described embodiment of the multiple-add operation multiplies together corresponding 16-bit data elements of Source1 and Source2 generating four 32-bit intermediate results. These 32-bit intermediate results are summed by pairs producing two 32-bit results that are packed into their respective elements of a packed result. As further described later, alternative embodiment may vary the number of bits in the data elements, intermediate results, and results. In addition, alternative embodiment may vary the number of data elements used, the number of intermediate results generated, and the number of data elements in the resulting packed data.

COMPUTER SYSTEM

[0023] Figure 1 illustrates an exemplary computer system 100 according to one embodiment of the invention. Computer system 100 includes a bus 101, or other communications hardware and software, for communicating information, and a processor 109 coupled with bus 101 for processing information. Processor 109 represents a central processing unit of any type of architecture, including a CISC or RISC type architecture. Computer system 100 further includes a random access memory (RAM) or other dynamic storage device (referred to as main memory 104), coupled to bus 101 for storing information and instructions to be executed by processor 109. Main memory 104 also may be used for storing temporary variables or other intermediate information during execution of instructions by processor 109. Computer system 100 also includes a read only memory (ROM) 106, and/or other static storage device, coupled to bus 101 for storing static information and instructions for processor 109. Data storage device 107 is coupled to bus 101 for storing information and instructions.

[0024] Figure 1 also illustrates that processor 109 includes an execution unit 130, a multiply-add unit 145, a register file 150, a cache 160, a decoder 165, and an internal bus 170. Of course, processor 109 contains additional circuitry which is not necessary to understanding the invention.

[0025] Execution unit 130 is used for executing instructions received by processor 109. In addition to recognizing instructions typically implemented in general purpose processors, execution unit 130 recognizes instructions 142 in packed instruction set 140 for performing operations on packed data formats. Packed instruction set 140 includes instructions for supporting multiply-add operations. In addition, packed instruction set 140 may also include instructions for supporting a pack operation, an unpack operation, a packed add operation, a packed subtract operation, a packed multiply operation, a packed shift operation, a packed compare operation, a population count operation, and a set of packed logical operations (including packed AND, packed ANDNOT, packed OR, and packed XOR). Execution unit 130 further includes the multiply-add unit 145 for performing multiply-add operations.

[0026] Execution unit 130 is coupled to register file 150 by internal bus 170. Register file 150 represents a storage area on processor 109 for storing information, including data. Execution unit 130 is further coupled to cache 160 and decoder 165. Cache 160 is used to cache data and/or control signals from, for example, main memory 104. Decoder 165 is used for decoding instructions received by processor 109 into control signals and/or microcode entry points. In response to these control signals and/or microcode entry points, execution unit 130 performs the appropriate operations. For example, if an add instruction is received, decoder 165 causes execution unit 130 to perform the required addition. Decoder 165 may be implemented using any number of different mechanisms (e.g., a look-up table, a hardware implementation, a PLA, etc.). Thus, while the execution of the various instructions by the decoder and execution unit is represented by a series of if/then statements, it is understood that the execution of an instruction does not require a serial processing of these if/then statements. Rather, any mechanism for logically performing this if/then processing is considered to be within the scope of the invention.

[0027] Figure 1 additionally shows a data storage device 107, such as a magnetic disk or optical disk, and its corresponding disk drive, can be coupled to computer system 100. Computer system 100 can also be coupled via bus 101 to a display device 121 for displaying information to a computer user. Display device 121 can include a frame buffer, specialized graphics rendering devices, a cathode ray tube (CRT), and/or a flat panel display. An alphanumeric input device 122, including alphanumeric and other keys, is typically coupled to bus 101 for communicating information and command selections to processor 109. Another type of user input device is cursor control 123, such as a mouse, a trackball, a pen, a touch screen, or cursor direction keys for communicating direction information and command selections to processor 109, and for controlling cursor movement on display device 121. This input device typically has two degrees of freedom in two axes, a first axis (e.g., x) and a second axis (e.g., y), which allows the device to specify positions in a plane. However, this invention should not be limited to input devices with only two degrees of freedom.

[0028] Another device which may be coupled to bus 101 is a hard copy device 124 which may be used for printing instructions, data, or other information on a medium such as paper, film, or similar types of media. Additionally, computer system 100 can be coupled to a device for sound recording, and/or playback 125, such as an audio digitizer coupled to a microphone for recording information. Further, the device may include a speaker which is coupled to a digital to analog (D/A) converter for playing back the digitized sounds.

[0029] Also, computer system 100 can be a terminal in a computer network (e.g., a LAN). Computer system 100 would then be a computer subsystem of a computer network. Computer system 100 optionally includes video digitizing device 126. Video digitizing device 126 can be used to capture video images that can be transmitted to others on the computer network.

[0030] In one embodiment, the processor 109 additionally supports an instruction set which is compatible with the x86 instruction set used by existing processors (such as the Pentium® processor) manufactured by Intel Corporation of Santa Clara, California. Thus, in one embodiment, processor 109 supports all the operations supported in the IA™ - Intel Architecture, as defined by Intel Corporation of Santa Clara, California (see Microprocessors, Intel Data Books volume 1 and volume 2, 1992 and 1993, available from Intel of Santa Clara, California). As a result, processor 109 can support existing x86 operations in addition to the operations of the invention. While the invention is described as being incorporated into an x86 based instruction set, alternative embodiments could incorporate the invention into other instruction sets. For example, the invention could be incorporated into a 64-bit processor using a new instruction set.

[0031] Figure 2 illustrates the register file of the processor according to one embodiment of the invention. The register file 150 is used for storing information, including control/status information, integer data, floating point data, and packed data. In the embodiment shown in Figure 2, the register file 150 includes integer registers 201, registers 209, status registers 208, and instruction pointer register 211. Status registers 208 indicate the status of processor 109. Instruction pointer register 211 Stores the address of the next instruction to be executed. Integer registers 201, registers 209, status registers 208, and instruction pointer register 211 are all coupled to internal bus 170. Any additional registers would also be coupled to internal bus 170.

[0032] In one embodiment, the registers 209 are used for both packed data and floating point data. In one such embodiment, the processor 109, at any given time, must treat the registers 209 as being either stack referenced floating point registers or non-stack referenced packed data registers. In this embodiment, a mechanism is included to allow the processor 109 to switch between operating on registers 209 as stack referenced floating point registers and non-stack referenced packed data registers. In another such embodiment, the processor 109 may simultaneously operate on registers 209 as non-stack referenced floating point and packed data registers. As another example, in another embodiment, these same registers may be used for storing integer data.

[0033] Of course, alternative embodiments may be implemented to contain more or less sets of registers. For example, an alternative embodiment may include a separate set of floating point registers for storing floating point data. As another example, an alternative embodiment may including a first set of registers, each for storing control/status information, and a second set of registers, each capable of storing integer, floating point, and packed data. As a matter

of clarity, the registers of an embodiment should not be limited in meaning to a particular type of circuit. Rather, a register of an embodiment need only be capable of storing and providing data, and performing the functions described herein.

[0034] The various sets of registers (e.g., the integer registers 201, the registers 209) may be implemented to include different numbers of registers and/or to different size registers. For example, in one embodiment, the integer registers 201 are implemented to store thirty-two bits, while the registers 209 are implemented to store eighty bits (all eighty bits are used for storing floating point data, while only sixty-four are used for packed data). In addition, registers 209 contains eight registers, R_0 212a through R_7 212h. R_1 212a, R_2 212b and R_3 212c are examples of individual registers in registers 209. Thirty-two bits of a register in registers 209 can be moved into an integer register in integer registers 201. Similarly, a value in an integer register can be moved into thirty-two bits of a register in registers 209. In another embodiment, the integer registers 201 each contain 64 bits, and 64 bits of data may be moved between the integer register 201 and the registers 209.

[0035] Figure 3 is a flow diagram illustrating the general steps are used by the processor to manipulate data according to one embodiment of the invention. That is, Figure 3 illustrates the steps followed by processor 109 while performing an operation on packed data, performing an operation on unpacked data, or performing some other operation. For example, such operations include a load operation to load a register in register file 150 with data from cache 160, main memory 104, read only memory (ROM) 106, or data storage device 107.

[0036] At step 301, the decoder 165 receives a control signal from either the cache 160 or bus 101. Decoder 165 decodes the control signal to determine the operations to be performed.

[0037] At step 302, Decoder 165 accesses the register file 150, or a location in memory. Registers in the register file 150, or memory locations in the memory, are accessed depending on the register address specified in the control signal. For example, for an operation on packed data, the control signal can include SRC1. SRC2 and DEST register addresses. SRC1 is the address of the first source register. SRC2 is the address of the second source register. In some cases, the SRC2 address is optional as not all operations require two source addresses. If the SRC2 address is not required for an operation, then only the SRC1 address is used. DEST is the address of the destination register where the result data is stored. In one embodiment, SRC1 or SRC2 is also used as DEST. SRC1, SRC2 and DEST are described more fully in relation to Figure 6a and Figure 6b. The data stored in the corresponding registers is referred to as Source1, Source2, and Result respectively. Each of these data is sixty-four bits in length.

[0038] In another embodiment of the invention, any one, or all, of SRC1, SRC2 and DEST, can define a memory location in the addressable memory space of processor 109. For example, SRC1 may identify a memory location in main memory 104, while SRC2 identifies a first register in integer registers 201 and DEST identifies a second register in registers 209. For simplicity of the description herein, the invention will be described in relation to accessing the register file 150. However, these accesses could be made to memory instead.

[0039] At step 303, execution unit 130 is enabled to perform the operation on the accessed data. At step 304, the result is stored back into register file 150 according to requirements of the control signal.

DATA AND STORAGE FORMATS

[0040] Figure 4 illustrates packed data-types according to one embodiment of the invention. Three packed data formats are illustrated; packed byte 401, packed word 402, and packed doubleword 403. Packed byte, in one embodiment of the invention, is sixty-four bits long containing eight data elements. Each data element is one byte long. Generally, a data element is an individual piece of data that is stored in a single register (or memory location) with other data elements of the same length. In one embodiment of the invention, the number of data elements stored in a register is sixty-four bits divided by the length in bits of a data element.

[0041] Packed word 402 is sixty-four bits long and contains four word 402 data elements. Each word 402 data element contains sixteen bits of information.

[0042] Packed doubleword 403 is sixty-four bits long and contains two doubleword 403 data elements. Each doubleword 403 data element contains thirty-two bits of information.

[0043] Figure 5a through 5c illustrate the in-register packed data storage representation according to one embodiment of the invention. Unsigned packed byte in-register representation 510 illustrates the storage of an unsigned packed byte 401 in one of the registers R_0 212a through R_7 212h. Information for each byte data element is stored in bit seven through bit zero for byte zero, bit fifteen through bit eight for byte one, bit twenty-three through bit sixteen for byte two, bit thirty-one through bit twenty-four for byte three, bit thirty-nine through bit thirty-two for byte four, bit forty-seven through bit forty for byte five, bit fifty-five through bit forty-eight for byte six and bit sixty-three through bit fifty-six for byte seven. Thus, all available bits are used in the register. This storage arrangement increases the storage efficiency of the processor. As well, with eight data elements accessed, one operation can now be performed on eight data elements simultaneously. Signed packed byte in-register representation 511 illustrates the storage of a signed packed byte 401. Note that the eighth bit of every byte data element is the sign indicator.

[0044] Unsigned packed word in-register representation 512 illustrates how word three through word zero are stored in one register of registers 209. Bit fifteen through bit zero contain the data element information for word zero, bit thirty-one through bit sixteen contain the information for data element word one, bit forty-seven through bit thirty-two contain the information for data element word two and bit sixty-three through bit forty-eight contain the information for data element word three. Signed packed word in-register representation 513 is similar to the unsigned packed word in-register representation 512. Note that the sixteenth bit of each word data element is the sign indicator.

[0045] Unsigned packed doubleword in-register representation 514 shows how registers 209 store two doubleword data elements. Doubleword zero is stored in bit thirty-one through bit zero of the register. Doubleword one is stored in bit sixty-three through bit thirty-two of the register. Signed packed doubleword in-register representation 515 is similar to unsigned packed doubleword in-register representation 514. Note that the necessary sign bit is the thirty-second bit of the doubleword data element.

[0046] As mentioned previously, registers 209 may be used for both packed data and floating point data. In this embodiment of the invention. the individual programming processor 109 may be required to track whether an addressed register, R_0 212a for example, is storing packed data or floating point data. In an alternative embodiment, processor 109 could track the type of data stored in individual registers of registers 209. This alternative embodiment could then generate errors if, for example, a packed addition operation were attempted on floating point data.

CONTROL SIGNAL FORMATS

[0047] The following describes one embodiment of the control signal formats used by processor 109 to manipulate packed data. In one embodiment of the invention, control signals are represented as thirty-two bits. Decoder 165 may receive the control signal from bus 101. In another embodiment, decoder 165 can also receive such control signals from cache 160.

[0048] Figure 6a illustrates a control signal format for indicating the use of packed data according to one embodiment of the invention. Operation field OP 601, bit thirty-one through bit twenty-six, provides information about the operation to be performed by processor 109; for example, packed addition, etc.. SRC1 602, bit twenty-five through twenty, provides the source register address of a register in registers 209. This source register contains the first packed data, Source1, to be used in the execution of the control signal. Similarly, SRC2 603, bit nineteen through bit fourteen, contains the address of a register in registers 209. This second source register contains the packed data, Source2, to be used during execution of the operation. DEST 605, bit five through bit zero, contains the address of a register in registers 209. This destination register will store the result packed data, Result, of the packed data operation.

[0049] Control bits SZ 610, bit twelve and bit thirteen, indicates the length of the data elements in the first and second packed data source registers. If SZ 610 equals 01_2, then the packed data is formatted as packed byte 401. If SZ 610 equals 10_2, then the packed data is formatted as packed word 402. SZ 610 equaling 00_2 or 11_2 is reserved, however, in another embodiment, one of these values could be used to indicate packed doubleword 403.

[0050] Control bit T 611, bit eleven, indicates whether the operation is to be carried out with saturate mode. If T 611 equals one, then a saturating operation is performed. If T 611 equals zero, then a non-saturating operation is performed. Saturating operations will be described later.

[0051] Control bit S 612, bit ten, indicates the use of a signed operation. If S 612 equals one, then a signed operation is performed. If S 612 equals zero, then an unsigned operation is performed.

[0052] Figure 6b illustrates a second control signal format for indicating the use of packed data according to one embodiment of the invention. This format corresponds with the general integer opcode format described in the "Pentium Processor Family User's Manual," available from Intel Corporation, Literature Sales, P.O. Box 7641, Mt. prospect, IL, 60056-7641. Note that OP 601, SZ 610, T 611, and S 612 are all combined into one large field. For some control signals, bits three through five are SRC1 602. In one embodiment, where there is a SRC1 602 address, then bits three through five also correspond to DEST 605. In an alternate embodiment, where there is a SRC2 603 address, then bits zero through two also correspond to DEST 605. For other control signals, like a packed shift immediate operation, bits three through five represent an extension to the opcode field. In one embodiment, this extension allows a programmer to include an immediate value with the control signal, such as a shift count value. In one embodiment, the immediate value follows the control signal. This is described in more detail in the "Pentium Processor Family User's Manual," in appendix F, pages F-1 through F-3. Bits zero through two represent SRC2 603. This general format allows register to register, memory to register, register by memory, register by register, register by immediate, register to memory addressing. Also, in one embodiment, this general format can support integer register to register, and register to integer register addressing.

DESCRIPTION OF SATURATE/UNSATURATE

[0053] As mentioned previously, T 611 indicates whether operations optionally saturate. Where the result of an op-

eration, with saturate enabled, overflows or underflows the range of the data, the result will be clamped. Clamping means setting the result to a maximum or minimum value should a result exceed the range's maximum or minimum value. In the case of underflow, saturation clamps the result to the lowest value in the range and in the case of overflow, to the highest value. The allowable range for each data format is shown in Table 4.

Table 4

Data Format	Minimum Value	Maximum Value
Unsigned Byte	0	255
Signed Byte	-128	127
Unsigned Word	0	65535
Signed Word	-32768	32767
Unsigned Doubleword	0	2^{64}-1
Signed Doubleword	-2^{63}	2^{63}-1

[0054] As mentioned above, T 611 indicates whether saturating operations are being performed. Therefore, using the unsigned byte data format, if an operation's result = 258 and saturation was enabled, then the result would be clamped to 255 before being stored into the operation's destination register. Similarly, if an operation's result = -32999 and processor 109 used signed word data format with saturation enabled, then the result would be clamped to -32768 before being stored into the operation's destination register.

MULTIPLY-ADD OPERATION

[0055] In one embodiment of the invention, the SRC1 register contains packed data (Source1), the SRC2 register contains packed data (Source2), and the DEST register will contain the result (Result) of performing the multiply-add operation on Source1 and Source2. In the first step of the multiply-add operation, Source1 will have each data element independently multiplied by the respective data element of Source2 to generate a set of respective intermediate results. These intermediate results are summed by pairs to generate the Result for the multiply-add operation.

[0056] In one embodiment of the invention, the multiply-add operation operates on signed packed data and truncate the results to avoid any overflows. In addition, the operation operates on packed word data and the Result is a packed double word. However, alternative embodiments could support the operation for other packed data types.

[0057] Figure 7 is a flow diagram illustrating the steps for performing multiply-add operations on packed data according to one embodiment of the invention.

[0058] At step 701, decoder 165 decodes the control signal received by processor 109. Thus, decoder 165 decodes: the operation code for a multiply-add operation.

[0059] At step 702, via internal bus 170, decoder 165 accesses registers 209 in register file 150 given the SRC1 602 and SRC2 603 addresses. Registers 209 provide execution unit 130 with the packed data stored in the SRC1 602 register (Source1), and the packed data stored in SRC2 603 register (Source2). That is, registers 209 communicate the packed data to execution unit 130 via internal bus 170.

[0060] At step 703, decoder 165 enables the multiply-add unit 145 of the execution unit 130 to perform the instruction. In step 714, the following is performed. Source1 bits fifteen through zero are multiplied by Source2 bits fifteen through zero generating a first 32-bit intermediate result (Intermediate Result 1). Sourcel bits thirty-one through sixteen are multiplied by Source2 bits thirty-one through sixteen generating a second 32-bit intermediate result (Intermediate Result 2). Source1 bits forty-seven through thirty-two are multiplied by Source2 bits forty-seven through thirty-two generating a third 32-bit intermediate result (Intermediate Result 3). Source1 bits sixty-three through forty-eight are multiplied by Source2 bits sixty-three through forty-eight generating a fourth 32-bit intermediate result (Intemmediate Result 4). Intermediate Result 1 is added to Intermediate Result 2 generating Result bits thirty-one through 0, and Intermediate Result 3 is added to Intermediate Result 4 generating Result bits sixty-three through thirty-two.

[0061] Different embodiments may perform the multiplies and adds serially, in parallel, or in some combination of serial and parallel operations.

[0062] At step 720, the Result is stored in the DEST register.

PACKED DATA MULTIPLY-ADD CIRCUIT

[0063] In one embodiment, the multiply-add operations can execute on multiple data elements in the same number

of clock cycles as a single multiply on unpacked data. To achieve execution in the same number of clock cycles, parallelism is used. That is, registers are simultaneously instructed to perform the multiply-add operations on the data elements.

[0064] In summary, Figure 8 illustrates a circuit for performing multiply-add operations on packed data according to one embodiment of the invention. Operation control 800 processes the control signal for the multiply-add instructions. Operation control 800 outputs signals on Enable 880 to control Packed multiply-adder 801.

[0065] Packed multiply-adder 801 has the following inputs: Source1[63:0] 831, Source2[63:0] 833 and Enable 880. Packed multiply-adder 801 includes four 16x16 multiplier circuits: 16x16 multiplier A 810, 16x16 multiplier B 811, 16x16 multiplier C 812 and 16x16 multiplier D 813. 16x16 multiplier A 810 has as inputs Source1[15:0] and Source2[15:0]. 16x16 multiplier B 811 has as inputs Source1[31:16] and Source2[31:16]. 16x16 multiplier C 812 has as inputs Source1 [47:32] and Source2[47:32]. 16x16 multiplier D 813 has as inputs Source1[63:48] and Source2[63:48]. The 32-bit intermediate results generated by 16x16 multiplier A 810 and 16x16 multiplier B 811 are received by adder 1350, while the 32-bit intermediate results generated by 16x16 multiplier C 812 and 16x16 multiplier D 813 are received by adder 851.

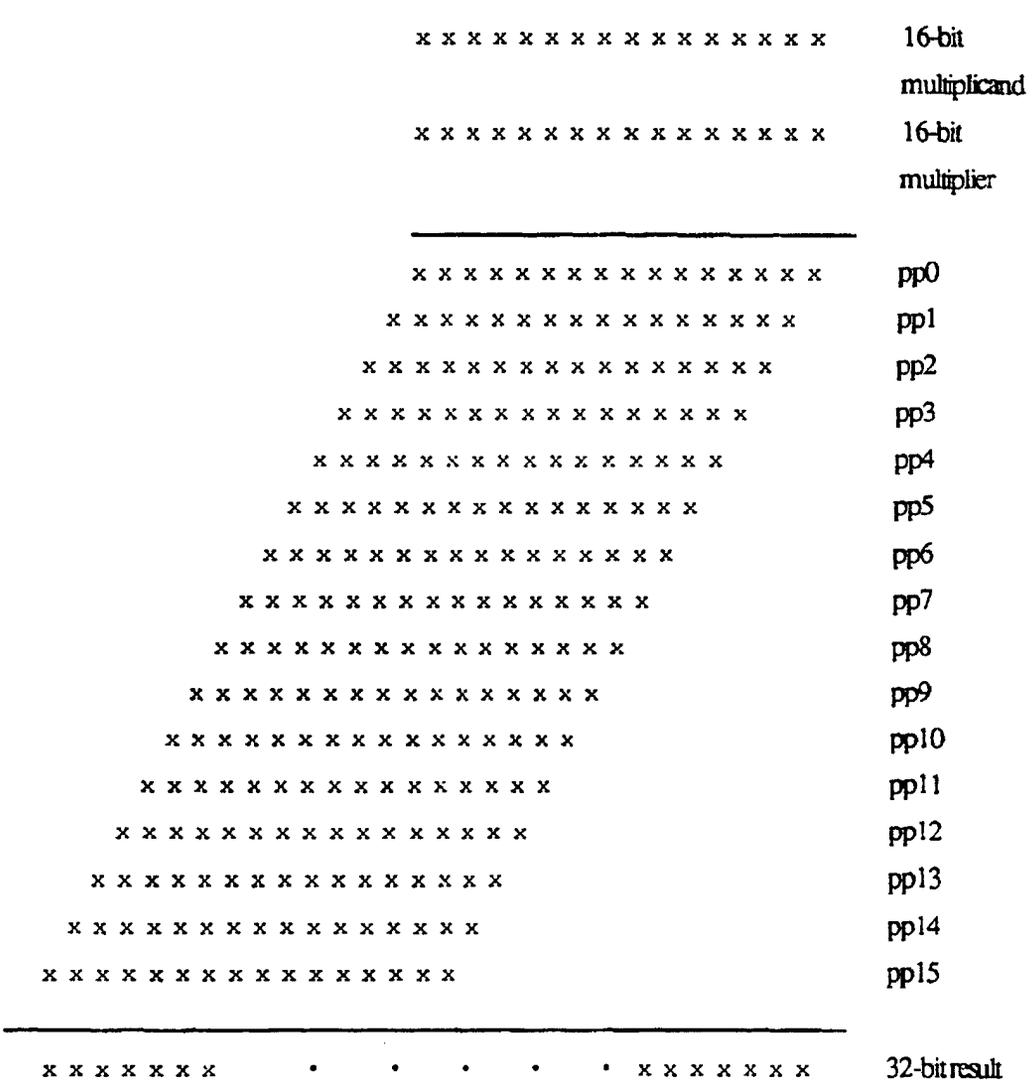
[0066] The adder 850 and adder 851 add their respective 32-bit inputs. The output of adder 850 (i.e., Result bits 31 through zero of the Result) and the output of adder 851 (i.e., bits 63 through 32 of the Result) are combined into the 64-bit Result and communicated to Result Register 871.

[0067] In one embodiment, each of adder 851 and adder 850 is composed of a 32-bit adder with the appropriate propagation delays. However, alternative embodiments could implement adder 851 and adder 850 in any number of ways.

[0068] Performing the equivalent of this multiply-add instruction using the prior art DSP processor described with reference to Table 1 requires one instruction to zero the accumulation value and four multiply accumulate instructions. Performing the equivalent of this multiply-add instruction using the prior art DSP processor described with reference to Table 2 requires one instruction to zero the accumulation value and 2-accumulate instructions.

[0069] In one embodiment of the multiply-add unit 145 of the present invention, each 16-bit multiplier used for the packed multiplication operations in the present invention is implemented using a 2-bit Booth algorithm. The main purpose of Booth algorithm in multipliers is to reduce the number of partial products to be summed. Fewer partial products consequently reduces the hardware and the area requirement for the multiplier. Table 5 below describes a common 16-bit multiplication process where 16 partial products are generated. Each partial product is shifted to the left by one bit and contains either all "0" terms or the exact replica of the multiplicand, depending on whether the respective bit of the multiplier is a "1" or a "0". A 32-bit result is generated by summing all 16 partial products slice by slice.

Table 5



[0070] On the other hand, the 2-bit Booth multiplier shown below in table 6 operates differently. In this case there are a total of 8 partial products and each partial product is 17 bits long. Every partial product has its own Booth encoder which dictates what the contents of the respective partial product. In alternative embodiments, other Booth encoder schemes could be used to select partial products.

[0071] A typical 2-bit Booth encoder has five outputs, which are for zero, plus 1, plus 2, minus 1, and minus 2 operations. Its truth table is given below in Table 6.

Table 6

BOOTH = -2yk + yk + yk - 1			
yk+1	YK	yk-1	BOOTH
0	0	0	0 zero (Z)
0 0 1			1 plusl (P1)
0 1 0			1 plusl (P2)
0	1	1	2 plus2 (M2)
1	0	0	-2 minus2 (M2)

Table 6 (continued)

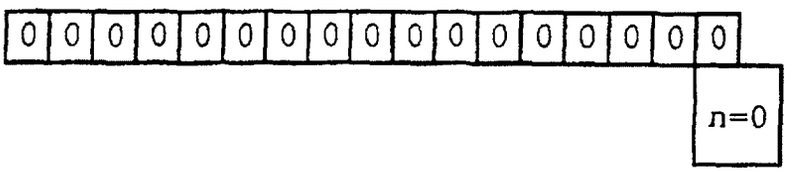
BOOTH = -2yk + yk + yk - 1			
yk+1	YK	yk-1	BOOTH
1	0	1	-1 minus1 (M1)
1	1	0	-1 minus1 (M1)
1	1	1	0 zero (Z)

[0072] As set forth in Table 6, yk+1, yk and yk-1 are the adjacent multiplier bits in descending order of significance. Table 7 below, further describes the form of the partial products according to the Booth encoder outputs.

Table 7

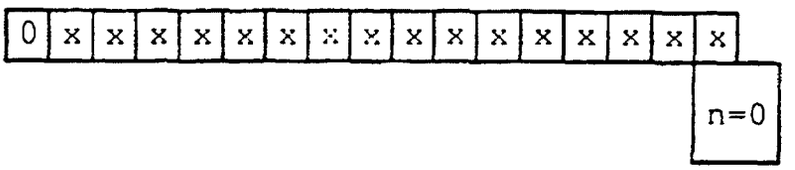
IF ZERO=1 then

17-bit zero string



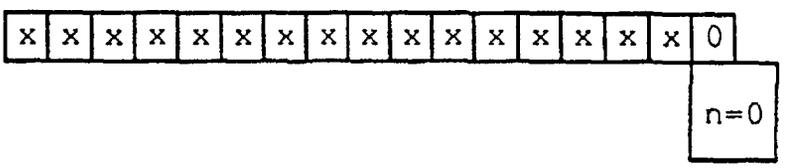
IF PLUS1=1 then

16-bit multiplicand



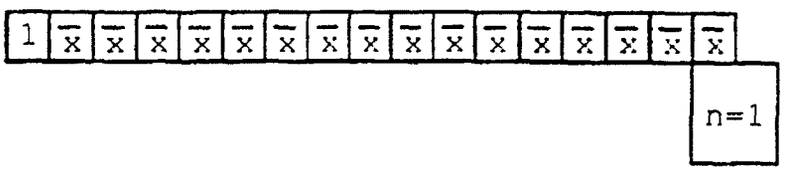
IF PLUS2=1 then

16-bit multiplicand left shifted by 1



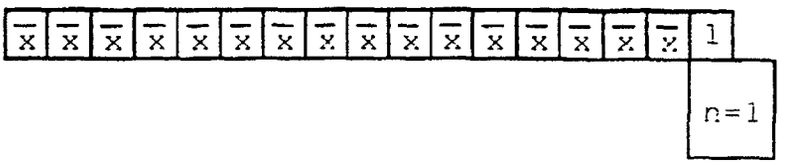
IF MINUS1=1 then

16-bit multiplicand complemented



IF MINUS 2=1 then

16-bit multiplicand complemented and shifted left by 1



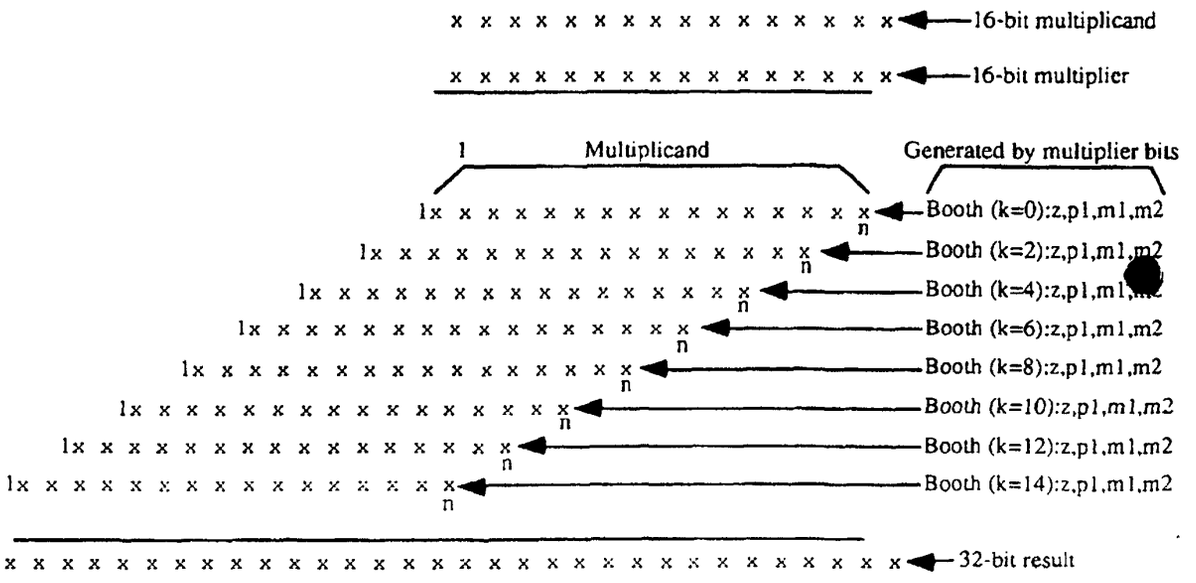
[0073] Instead of sign extending the partial products, a sign generate method is used to reduce the number of Carry Same Address (CSA's) needed for partial product reduction. Table 8 shows the sign generate method. The complement of the sign bit of a partial product is prepended to the partial product. Two one bits are then prepended to the complement of the sign bit.

Table 8

Sign-extension method	Sign-generate method
	1
S1 S1 S1 partial product 1	1 S1 partial product
1	
S2 S2 partial product 2	1 S2 partial product 2
partial product 3	1 S3 partial product 3

[0074] Table 9 below shows the 16-bit multiplication process used in one embodiment of the present invention using the 2-bit Booth encoder and the sign generate method to generate 8 partial products.

Table 9



[0075] In one embodiment, each Booth encoder contains 3 adjacent multiplier bits (y0 is the least significant bit and y15 is the most significant bit of the multiplier bits). The form of the partial product is defined by 17 selectors and a negate circuit connected to a particular Booth encoder. The data bits of the selectors contain 2 adjacent multiplier bits, their compliments and Vcc (for zero operation). The selected outputs then go through a maze of partial product reduction tree, commonly referred to as a Wallace tree.

[0076] Once the partial products are all selected then the summation of partial products begins. The Wallace tree is made out of full adders and half adders. Figures 9a-9e illustrate the Wallace tree performing the partial product summation and reduction for one embodiment of the present invention, as implemented by each of the four 16-bit multipliers in the multiply-add unit 145. As illustrated, 8 partial products are reduced to 6 partial products, then to 4 partial products, then to 3 partial products, and then finally to 2 partial products.

[0077] More specifically, as illustrated in Figures 9a-9e, the subscript numbers in each row represent bits of a partial product (a_{se15}, a_{s15}, a_{14} - a_0). Each row represents a separate partial product. Negh - nega represent the + 1 part of

a 2's complement, for each partial product. As a result, if a certain Booth encoded bit of the multiplier is negative, that corresponding partial product's "neg" bit is 1, set forth in the next row.

[0078] In addition, as further illustrated in Figures 9a-9e, $S_{<position><adder\ number>}$ represents the sum portion of a carry-save adder. $_{<adder\ number>}$ indicates to which row of adders sum belongs. Adders are numbered from top to bottom of Figures 9a-9e. $_{<position>}$ indicates which bit position (0...31) this adder operates. For example, S_{42} is the sum of a carry-save adder 2 that corresponds to bit position 4.

[0079] $C_{<position><level>}$ represents the carry portion of a carry-save adder. $_{<level>}$ indicates the respective row of adders for the carry. $_{<position>}$ indicates which bit position (0...31) this adder operates. A carry-save adder can be a full adder, or a half-adder. A full adder adds 3 bits together. A half-adder adds 2 bits.

[0080] Figures 9a-9e further illustrate implementation of the Sign-Generate method as previously described. As illustrated, the Sign-Generate method creates a '1 on bit position 16 in row above the first partial product row. In the last partial product row, if the most significant Booth encoding number is negative, a 1 is created in a row below because the partial product is 2's complemented. This process would typically require 10 rows instead of 8. The more rows a multiplication has, the more hardware is needed to reduce the partial products into 2 numbers on which a carry-propagate adder can add.

[0081] Binary multiplication, however, is performed through addition. When two numbers are added, the order of the numbers is irrelevant. The result is the same regardless of which number is the first number. This principle is used throughout the Wallace tree to reduce the number of carry-save's needed. Specifically, in one embodiment, the 1 in bit position 16 of the first row in Figure 9a is moved down to bit position 16 in the last row of Figure 9, which also contains negh. As a result, less hardware is needed to do the reduction because fewer rows are used.

[0082] Figure 9b illustrates the result of the first level of reduction. The boxes indicate which bits are operated on by carry-save adders. A number of bits are moved around to fit everything in 6 rows. For example, bit d_{se15} is moved to the first row. $c_{se}15$ is moved to a fifth row. A bit, however, should only be moved up or down in the same bit position. From Figure 9b to Figure 9c, the number of rows is reduced to 4. From Figure 9c to Figure 9e, the number of rows is reduced to 3. Finally, one more row of carry-save adders reduces the number of rows to 2, as shown in Figure 9e.

[0083] Figures 10a - 10f illustrate one embodiment of a circuit, comprised of full adders and half adders, implementing the Wallace tree diagram illustrated in Figures 9a-9e.

[0084] Figure 11 illustrates one embodiment of a block diagram of the unit. There are 2 identical 16-bit multipliers illustrated. The multipliers could perform the multiply-add operation on either the 0-31 bit or the 32-63 bits. An additional 2 16-bit multipliers would also be provided, very similar in structure to the multipliers illustrated, to complete the multiply-add unit 145. As illustrated, each multiplier accomplishes a 16-bit multiplication resulting in 2 partial products in 1.5 clock cycles. In the next half clock cycle, which is the low phase of the 34th clock, the 4 partial products generated by multipliers 1110 and 1120 are summed again by a 4:2 CSA 1130. The control signal 1150 selects either the partial product of the multiplier 1110 or the partial product generated at the output of the 4:2 CSA 1130 (Sumres and coutres). The selected data is latched along with the partial products of multiplier 1120. At the high phase of the 35th clock, CPA 1140 generates a 32-bit result by adding the resultant partial products. The final partitioning of the 32-bit sum is accomplished by a mux-latch outside of the fub and the selected data is written back at the low phase of the 35th clock cycle.

[0085] In one embodiment, the multiply-add unit 145 operates with a 3 latency-1 throughput rule. Otherwise stated, the unit 145 requires 3 clock cycles to complete its task every time an unpiped data stream is introduced to the inputs. All the inputs of the unit 145 are buffered, which offers very small capacitance to the outside world.

[0086] Even though the data is available at the inputs of unit 145 at the beginning of 32L, the data may not be valid until 33H begins. Therefore, the multiplication is assumed to start at the beginning of 33H. During 33H the valid and stable data on the multiplicand inputs flow through the delay elements and get latched by latches 1160 and 1170, respectively. At the same time, the data on the multiplier inputs propagate through the input buffers and the Booth encoders 1165 an 1175 and get latched by latches 1180 and 1182. At this point, the data on both multiplier and multiplicand paths are perfectly synchronized with each other. In 331, the data go through the bit selector array and a set of full adders, which forms the first part of the Wallace tree and becomes valid before the setup time for latches 1180 and 1182. The number of partial products at this point is reduced from 8 to 4. In 34H, the data goes through another set of full adders which constitute the remainder of the Wallace tree and become valid and stable at the end of 34H before getting latched by latches 1184 and 1186.

[0087] As previously explained, during 34L the data goes through 4:2 compressor 1130 (two full adders in series) and a 2-1 mux 1135 for the final partial product selection. The data becomes valid at the end of 34L before getting latched by latch 1190. During 35H, the two partial products at the latch 1190 output are finally reduced to one sum vector. This sum vector gets portioned and latched by a mux-latch boundary, latch 1195, outside of unit 145. In 35L, the data is available for write-back operation.

[0088] As previously described, the previous multiply accumulate instructions always add the results of their multiplications to an accumulation value. This accumulation value becomes a bottleneck for performing operations other

than multiplying and accumulating (e.g., the accumulation value must be cleared each time a new set of operations is required which do not require the previous accumulation value). This accumulation value also becomes a bottleneck if operations, such as rounding, need to be performed before accumulation.

[0089] In contrast, the disclosed multiply-add operation does not carry forward an accumulation value. As a result, these instructions are easier to use in a wider variety of algorithms. In addition, software pipelining can be used to achieve comparable throughput. To illustrate the versatility of the multiply-add instruction, several example multimedia algorithms are described below. Some of these multimedia algorithms use additional packed data instructions. The operation of these additional packed data instructions are shown in relation to the described algorithms. Of course, other packed data instructions could be used. In addition, a number of steps requiring the use of general purpose processor instructions to manage data movement, looping, and conditional branching have been omitted in the following examples.

1) Multiplication of Complex Numbers

[0090] The disclosed multiply-add instruction can be used to multiply two complex numbers in a single instruction as shown in Table 10a. As previously described, the multiplication of two complex number (e.g., r_1 i_1 and r_2 i_2) is performed according to the following equation:

$$\text{Real Component} = r_1 \bullet r_2 - i_1 \bullet i_2$$

$$\text{Imaginary Component} = r_1 \bullet i_2 + r_2 \bullet i_1$$

If this instruction is implemented to be completed every clock cycle, the invention can multiply two complex numbers every clock cycle.

Table 10a

Multiply-Add Source1, Source2

r_1	i_2	r_1	i_1	Source1
r_2	$-i_2$	i_2	r_2	Source2
=				
Real Component: $r_1r_2-i_1i_2$		Imaginary Component: $r_1i_2+r_2i_1$		Result 1

[0091] As another example, Table 10b shows the instructions used to multiply together three complex numbers.

Multiply-Add Source1, Source2

r_1	i_1	r_1	i_1	Source1
r_2	$-i_2$	i_2	r_2	Source2
=				
Real Component1: $r_1r_2-i_1i_2$		Imaginary Component1: $r_1i_2+r_2i_1$		Result1

Packed Shift Right Source1, Source2

Real Component1			Imaginary Component1	Result1
16				
=				
	Real Component1		Imaginary Component1	Result2

Table 10b

Pack Result2, Result2

	Real Component1		Imaginary Component1	Result2
	Real Component1		Imaginary Component1	Result2
=				
Real Component1	Imaginary Component1	Real Component1	Imaginary Component1	Result3

Multiply-Add Result3, Source3

Real Component1: r1r2-i1i2	Imaginary Component1: r1i2+r2i1	Real Component1: r1r2-i1i2	Imaginary Component1: r1i2+r2i1	Result3
r3	-i3	i3	r3	Source3
=				
Real Component2		Imaginary Component2		Result4

2) Multiply Accumulation Operations

[0092] The disclosed multiply-add instructions can also be used to multiply and accumulate values. For example, two sets of four data elements (A_{1-4} and B_{1-4}) may be multiplied and accumulated as shown below in Table 11. In one embodiment, each of the instructions shown in Table 6 is implemented to complete each clock cycle.

Table 11

Multiply-Add Source1, Source2

0	0	A_1	A_2	Source1
0	0	B_1	B_2	Source2
		=		
0		$A_1B_1+A_2B_2$		Result1

Multiply-Add Source3, Source4

0	0	A_3	A_4	Source3
0	0	B_3	B_4	Source4
		=		
0		$A_3A_4+B_3B_4$		Result2

Unpacked Add Result1, Result2

0	$A_1B_1+A_2B_2$	Result1
0	$A_3A_4+B_3B_4$	Result2
=		
0	$A_1B_1+A_2B_2+A_3A_4+B_3B_4$	Result3

[0093] If the number of data elements in each set exceeds 8 and is a multiple of 4, the multiplication and accumulation of these sets requires fewer instructions if performed as shown in Table 12 below.

Table 12

Multiply-Add Source1, Source2

A_1	A_2	A_3	A_4	Source1
B_1	B_2	B_3	B_4	Source2
=				
$A_1B_1+A_2B_2$		$A_3B_3+A_4B_4$		Result1

Multiply-Add Source3, Source4

A_5	A_6	A_7	A_8	Source3
B_5	B_6	B_7	B_8	Source4
=				
$A_5B_5+A_6B_6$		$A_7B_7+A_8B_8$		Result2

Packed Add Result1, Result2

$A_1B_1+A_2B_2$	$A_3B_3+A_4B_4$	Result1
$A_5B_5+A_6B_6$	$A_7B_7+A_8B_8$	Result2
=		
$A_1B_1+A_2B_2+A_5B_5+A_6B_6$	$A_3B_3+A_4B_4+A_7B_7+A_8B_8$	Result3

Unpack High Result3, Source5

$A_1B_1+A_2B_2+A_5B_5+A_6B_6$	$A_3B_3+A_4B_4+A_7B_7+A_8B_8$	Result3
0	0	Source5
=		
0	$A_1B_1+A_2B_2+A_5B_5+A_6B_6$	Result4

Unpack Low Result3, Source5

$A_1B_1+A_2B_2+A_5B_5+A_6B_6$	$A_3B_3+A_4B_4+A_7B_7+A_8B_8$	Result3
0	0	Source5
=		
0	$A_3B_3+A_4B_4+A_7B_7+A_8B_8$	Result5

Packed Add Result4, Result5

0	$A_1B_1+A_2B_2+A_5B_5+A_6B_6$	Result4
0	$A_3B_3+A_4B_4+A_7B_7+A_8B_8$	Result5
=		
0	TOTAL	Result6

[0094] As another example, Table 13 shows the separate multiplication and accumulation of sets A and B and sets C and D, where each of these sets includes 2 data elements.

Table 13

Multiply-Add Source1, Source2

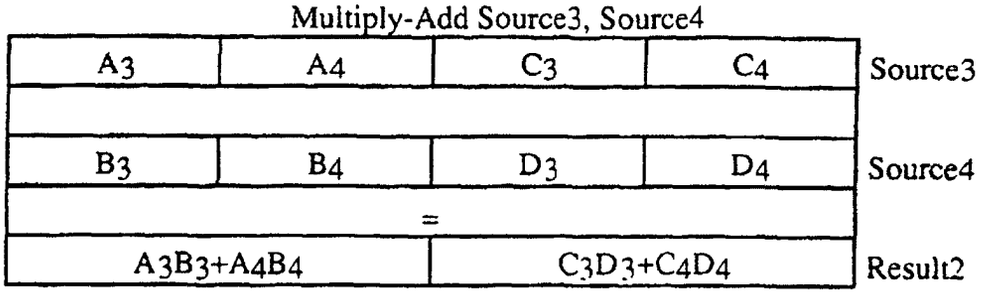
A_1	A_2	C_1	C_2	Source1
B_1	B_2	D_1	D_2	Source2
=				
$A_1B_1+A_2B_2$		$C_1D_1+C_2D_2$		Result1

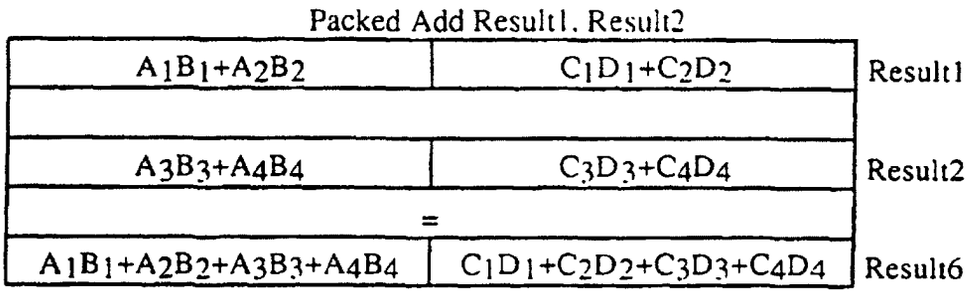
[0095] As another example, Table 14 shows the separate multiplication and accumulation of sets A and B and sets C and D, where each of these sets includes 4 data elements.

Table 14

Multiply-Add Source1, Source2

A_1	A_2	C_1	C_2	Source1
B_1	B_2	D_1	D_2	Source2
=				
$A_1B_1+A_2B_2$		$C_1D_1+C_2D_2$		Result1





3) Dot Product Algorithms

[0096] Dot product (also termed as inner product) is used in signal processing and matrix operations. For example, dot product is used when computing the product of matrices, digital filtering operations (such as FIR and IIR filtering), and computing correlation sequences. Since many speech compression algorithms (e.g., GSM, G.728, CELP, and VSELP) and Hi-Fi compression algorithms (e.g., MPEG and subband coding) make extensive use of digital filtering and correlation computations, increasing the performance of dot product increases the performance of these algorithms.

[0097] The dot product of two length N sequences A and B is defined as:

$$\text{Result} = \sum_{i=0}^{N-1} Ai \cdot Bi$$

[0098] Performing a dot product calculation makes extensive use of the multiply accumulate operation where corresponding elements of each of the sequences are multiplied together, and the results are accumulated to form the dot product result.

[0099] The dot product calculation can be performed using the multiply-add instruction. For example if the packed data type containing four sixteen-bit elements is used, the dot product calculation may be performed on two sequences each containing four values by:

1) accessing the four sixteen-bit values from the *A* sequence to generate Source1 using a move instruction;
2) accessing four sixteen-bit values from the *B* sequence to generate Source2 using a move instruction; and
3) performing multiplying and accumulating as previously described using a multiply-add, packed add, and shift instructions.

[0100] For vectors with more than just a few elements the method shown in Table 9 is used and the final results are added together at the end. Other supporting instructions include the packed OR and XOR instructions for initializing the accumulator register, the packed shift instruction for shifting off unwanted values at the final stage of computation.

Loop control operations are accomplished using instructions already existing in the instruction set of processor 109.

4) Discrete Cosign Transform Algorithms

[0101] Discrete Cosine Transform (DCT) is a well known function used in many signal processing algorithms. Video and image compression algorithms, in particular, make extensive use of this transform.

[0102] In image and video compression algorithms, DCT is used to transform a block of pixels from the spatial representation to the frequency representation. In the frequency representation, the picture information is divided into frequency components, some of which are more important than others. The compression algorithm selectively quantifies or discards the frequency components that do not adversely affect the reconstructed picture contents. In this manner, compression is achieved.

[0103] There are many implementations of the DCT, the most popular being some kind of fast transform method modeled based on the Fast Fourier Transform (FFT) computation flow. In the fast transform, an order N transform is broken down to a combination of order N/2 transforms and the result recombined. This decomposition can be carried out until the smallest order 2 transform is reached. This elementary 2 transform kernel is often referred to as the butterfly operation. The butterfly operation is expressed as follows:

$$X = a*x + b*y$$

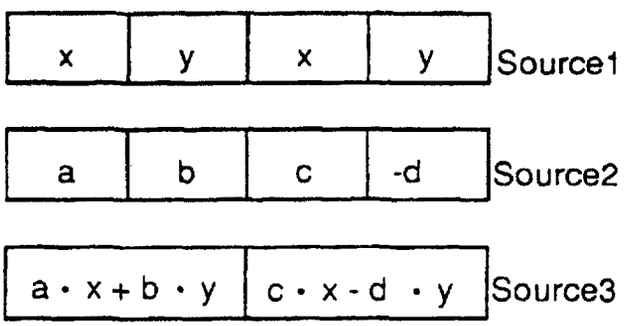
$$Y = c*x - d*y$$

where a, b, c and d are termed the coefficients, x and y are the input data, and X and Y are the transform output.

[0104] The multiply-add allows the DCT calculation to be performed using packed data in the following manner:

1) accessing the two 16-bit values representing x and y to generate Source1 (see Table 10 below) using the move and unpack instructions;
2) generating Source2 as shown in Table 10 below -- Note that Source2 may be reused over a number of butterfly operations; and
3) performing a multiply-add instruction using Source1 and Source2 to generate the Result (see Table 15 below).

Table 15



In some situations, the coefficients of the butterfly operation are 1. For these cases, the butterfly operation degenerates into just adds and subtracts that may be performed using the packed add and packed subtract instructions.

[0105] An IEEE document specifies the accuracy with which inverse DCT should be performed for video conferencing. (See, IEEE Circuits and Systems Society, "IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform," IEEE Std. 1180-1990, IEEE Inc. 345 East 47th St., NY, NY 10017, USA, March 18, 1991). The required accuracy is met by the disclosed multiply-add instruction because it uses 16-bit inputs to generate 32-bit outputs.

[0106] In this manner, the described multiply-add instruction can be used to improve the performance of a number of different algorithms, including algorithms that require the multiplication of complex numbers, algorithms that require transforms, and algorithms that require multiply accumulate operations. As a result, this multiply-add instruction can

be used in a general purpose processor to improve the performance of a greater number algorithms than the described previous instructions.

[0107] While the invention has been described in terms of several embodiments, those skilled in the art will recognise that the invention is not limited to the embodiments described. The method and apparatus of the invention can be practised with modification and alteration within the scope of the appended claims. The description is thus to be regarded as illustrative instead of limiting on the invention.

Claims

1. A processor (109) comprising:

a register file (150) to store a first packed data and a second packed data each containing four initial data elements, each of the initial data elements in the first packed data corresponding to a initial data element in the second packed data to create four pairs of initial data elements,
a decoder (165) to decode an instruction that specifies the first and second packed data as operands, and
a circuit (145) coupled to the register file and the decoder, the circuit in response to the instruction to, multiply together the corresponding initial data elements in the first and second packed data to generate corresponding intermediate data elements, the intermediate data elements being paired into two sets,

characterised in that the circuit (145) adds the intermediate data elements in each of the two sets to generate two result data elements, wherein each of the two result data elements provides a higher precision than the initial data elements, and store as a result of executing the single instruction the two result data elements in one register of the register file as a third packed data wherein the third packed data contains only the two result data elements.

2. The processor (109) of claim 1, wherein each of the four initial data elements include 16-bits of data, each of the two intermediate data elements include 32 bits of data, and each of the two result data elements include 32 bits of data.

3. The processor (109) of claim 1 wherein the circuit, in response to the instruction, simultaneously multiplies the corresponding initial data elements in the first and second packed data to generate corresponding intermediate data elements.

4. The processor (109) of claim 3, wherein the circuit, in response to the instruction, simultaneously multiplies in a same cycle the corresponding initial data elements in the first and second packed data to generate corresponding intermediate data elements.

5. The processor (109) of claim 5, wherein the circuit, in response to the instruction, simultaneously multiplies in a same clock cycle the corresponding initial data elements in the first and second packed data to generate corresponding intermediate data elements.

6. The processor (109) of claim 1, wherein the circuit (145) is a multiply-add circuit having a first (813), second (812), third (811), and fourth (810) multiplier, each the multiplier to receive a corresponding set of the data elements; the multiply-add circuit further including a first adder (851) coupled to the first multiplier and second multiplier, and a second adder (850) coupled to the third multiplier and fourth multiplier.

7. The processor (109) of claim 6, wherein the two result data elements of the third packed data contain twice as many bits as the data elements in the first and second packed data.

8. The processor (109) of claim 7 wherein the multiply-add circuit (145) includes a first Carry Save Adder (CSA) coupled to the first (813) and a second (812) multipliers, the first CSA to receive two partial products generated by the first multiplier (815) and receives two partial products generated by the second multiplier (812), the first CSA to generate a first set of two partial products, and a second (CSA) coupled to the third (817) and fourth multipliers (810), the second CSA to receive two partial products generated by the third multiplier (811) and receive two partial products generated by the fourth multiplier (810), the second CSA to generate a second set of two partial products.

9. The processor (109) of claim 8 wherein each multiplier performs at least a 16-bit multiplication resulting in two

partial products.

10. The processor (109) of claim 9 wherein the multipliers each select eight partial products and each the multiplier includes four levels of CSA's.

11. The processor (109) of claim 10, wherein each of the four levels of CSA's included in each multiplier include a first level reducing the eight partial products to six partial products, a second level reducing the six partial products to four partial products, a third level reducing four partial products to three partial products, and a fourth level reducing the three partial products to two partial products.

12. The processor (109) of claim 10, wherein the first adder (851) is coupled to the first CSA and the first adder (851) generates a summation of the two partial products generated by the first CSA; and the second adder (850) is coupled to the second CSA and the second adder (850) generates a summation of the two partial products generated by the second CSA.

13. The processor (109) of claim 8, wherein each the multiplier implements a 2-bit Booth encoding to select a set of eight partial products.

14. The processor (109) of claim 13, wherein each the multiplier implements a sign generate method to replace sign extending of the partial products.

15. The processor (109) of claim 14, wherein each the multiplier relocates an extra bit generated by the sign generation method to a corresponding vacant bit position in a separate partial product row.

Patentansprüche

1. Ein Prozessor (109), aufweisend:

einen Registersatz (150) zum Speichern erster gepackter Daten und zweiter gepackter Daten, die jeweils vier anfängliche Datenelemente enthalten, wobei jedes der anfänglichen Datenelemente in den ersten gepackten Daten mit einem anfänglichen Datenelement in den zweiten gepackten Daten korrespondiert, so daß vier Paare anfänglicher Datenelemente gebildet werden,
einen Decodierer (165) zum Decodieren eines Befehls, der die ersten und die zweiten gepackten Daten als Operanden spezifiziert, und
eine mit dem Registersatz und dem Decodierer gekoppelte Schaltung (145), wobei die Schaltung in Erwiderung des Befehls zum Multiplizieren der korrespondierenden anfänglichen Datenelemente in den ersten und den zweiten gepackten Daten dient, um zugehörige Zwischendatenelemente zu erzeugen, wobei die Zwischendatenelemente in zwei Sätzen gepaart werden,

dadurch gekennzeichnet, daß die Schaltung (145) die Zwischendatenelemente in jedem der beiden Sätze addiert, um zwei Ergebnisdatenelemente zu erzeugen, wobei jedes der beiden Ergebnisdatenelemente eine höhere Genauigkeit zur Verfügung stellt als die anfänglichen Datenelemente, und wobei im Ergebnis des Ausführens des einzelnen Befehls die beiden Ergebnisdatenelemente in einem Register des Registersatzes als dritte gepackte Daten gespeichert werden, wobei die dritten gepackten Daten nur die beiden Ergebnisdatenelemente enthalten.

2. Der Prozessor (109) nach Anspruch 1, wobei jedes der vier anfänglichen Datenelemente 16 Bits Daten enthält, wobei jedes der zwei Zwischendatenelemente 32 Bits Daten enthält und wobei jedes der zwei Ergebnisdatenelemente 32 Bits Daten enthält.

3. Der Prozessor (109) nach Anspruch 1, wobei die Schaltung in Erwiderung des Befehls gleichzeitig die korrespondierenden anfänglichen Datenelemente in den ersten und den zweiten gepackten Daten multipliziert, um zugehörige Zwischendatenelemente zu erzeugen.

4. Der Prozessor (109) nach Anspruch 3, wobei die Schaltung in Erwiderung des Befehls gleichzeitig in demselben Zyklus die korrespondierenden anfänglichen Datenelemente in den ersten und den zweiten gepackten Daten multipliziert, um zugehörige Zwischendatenelemente zu erzeugen.

5. Der Prozessor (109) nach Anspruch 5, wobei die Schaltung in Erwiderung des Befehls gleichzeitig in demselben Taktzyklus die korrespondierenden anfänglichen Datenelemente in den ersten und den zweiten gepackten Daten multipliziert, um zugehörige Zwischendatenelemente zu erzeugen.

6. Der Prozessor (109) nach Anspruch 1, wobei die Schaltung (145) eine Multiplizier-Addier-Schaltung mit einem ersten (813), einem zweiten (812), einem dritten (811) und einem vierten (810) Multiplizierer ist, wobei jeder der Multiplizierer einen zugehörigen Satz der Datenelemente empfängt; wobei die Multiplizier-Addier-Schaltung ferner einen ersten Addierer (851), der mit dem ersten Multiplizierer und dem zweiten Multiplizierer gekoppelt ist, und einen zweiten Addierer (850), der mit dem dritten Multiplizierer und dem vierten Multiplizierer gekoppelt ist, enthält.

7. Der Prozessor (109) nach Anspruch 6, wobei die beiden Ergebnisdatenelemente der dritten gepackten Daten doppelt so viele Bits wie die Datenelemente in den ersten und den zweiten gepackten Daten enthalten.

8. Der Prozessor (109) nach Anspruch 7, wobei die Multiplizier-Addier-Schaltung (145) einen mit dem ersten (813) und einem zweiten (812) Multiplizierer gekoppelten ersten Übertragssicherungsaddierer (CSA) enthält, wobei der erste CSA zwei von dem ersten Multiplizierer (815) erzeugte Partialprodukte empfängt und zwei von dem zweiten Multiplizierer (812) erzeugte Partialprodukte empfängt, wobei der erste CSA einen ersten Satz von zwei Partialprodukten erzeugt, und wobei die Multiplizier-Addier-Schaltung einen mit dem dritten (817) und dem vierten (810) Multiplizierer gekoppelten zweiten Übertragssicherungsaddierer (CSA) enthält, wobei der zweite CSA zwei von dem dritten Multiplizierer (811) erzeugte Partialprodukte empfängt und zwei von dem vierten Multiplizierer (810) erzeugte Partialprodukte empfängt, und wobei der zweite CSA einen zweiten Satz von zwei Partialprodukten erzeugt.

9. Der Prozessor (109) nach Anspruch 8, wobei jeder Multiplizierer wenigstens eine 16-Bit-Multiplikation ausführt, die zu zwei Partialprodukten führt.

10. Der Prozessor (109) nach Anspruch 9, wobei die Multiplizierer jeweils acht Partialprodukte auswählen und jeder der Multiplizierer vier Ebenen von CSAs enthält.

11. Der Prozessor (109) nach Anspruch 10, wobei jede der vier Ebenen der CSAs, die in jedem Multiplizierer enthalten sind, eine die acht Partialprodukte zu sechs Partialprodukten reduzierende erste Ebene, einen die sechs Partialprodukte zu vier Partialprodukten reduzierende zweite Ebene, eine vier Partialprodukte zu drei Partialprodukten reduzierende dritte Ebene und eine die drei Partialprodukte zu zwei Partialprodukten reduzierende vierte Ebene enthält.

12. Der Prozessor (109) nach Anspruch 10, wobei der erste Addierer (851) mit dem ersten CSA gekoppelt ist und der erste Addierer (851) eine Summation der zwei von dem ersten CSA erzeugten Partialprodukte erzeugt; und wobei der zweite Addierer (850) mit dem zweiten CSA gekoppelt ist und der zweite Addierer (850) eine Summation der zwei von dem zweiten CSA erzeugten Partialprodukte erzeugt.

13. Der Prozessor (109) nach Anspruch 8, wobei jeder der Multiplizierer eine 2-Bit-Booth-Codierung implementiert, um einen Satz von acht Partialprodukten auszuwählen.

14. Der Prozessor (109) nach Anspruch 13, wobei jeder der Multiplizierer ein Vorzeichenerzeugungsverfahren implementiert, um eine Vorzeichenerweiterung der Partialprodukte zu ersetzen.

15. Der Prozessor (109) nach Anspruch 14, wobei jeder der Multiplizierer ein Zusatzbit, das von dem Vorzeichenerzeugungsverfahren erzeugt wird, zu einer zugehörigen vakanten Bitposition in einer separaten partiellen Produktzeile verschiebt.

Revendications

1. Un processeur (109) comprenant:

   un fichier des registres (150) pour mémoriser une première donnée comprimée et une deuxième donnée comprimée comprenant chacune quatre éléments de donnée initiaux, chacun des éléments de donnée initiaux dans la première donnée comprimée correspondant à un élément de donnée initial dans la deuxième donnée

comprimée pour créer quatre paires d'éléments de donnée initiaux,
un décodeur (165) pour décoder une instruction qui spécifie les première et deuxième données comprimées comme des opérandes, et
un circuit (145) couplé au fichier des registres et au décodeur, le circuit, en réponse à l'instruction, pour multiplier ensemble les éléments de donnée initiaux correspondants dans les première et deuxième données comprimées pour générer des éléments de donnée intermédiaires correspondants, les éléments de donnée intermédiaires étant appariés en deux ensembles,

caractérisé en ce que le circuit (145) additionne les éléments de donnée intermédiaires dans chacun des deux ensembles pour engendrer deux éléments de donnée de résultat, dans lesquels chacun des deux éléments de donnée de résultat fournit une plus grande précision que les éléments de donnée initiaux et mémorise les deux éléments de donnée de résultat, comme résultat de l'exécution de l'instruction individuelle, dans un registre du fichier des registres comme une troisième donnée comprimée, où la troisième donnée comprimée contient seulement les deux éléments de donnée de résultat.

2. Le processeur (109) selon la revendication 1, dans lequel chacun des quatre éléments de donnée initiaux comprend 16 bits de donnée, chacun des deux éléments de donnée intermédiaires comprend 32 bits de donnée et chacun des deux éléments de donnée de résultat comprend 32 bits de donnée,

3. Le processeur (109) selon la revendication 1, dans lequel le circuit, en réponse à l'instruction, multiplie simultanément les éléments de donnée initiaux correspondants dans les première et deuxième données comprimées pour générer des éléments de donnée intermédiaires correspondants.

4. Le processeur (109) selon la revendication 3, dans lequel le circuit, en réponse à l'instruction, multiplie simultanément dans un même cycle les éléments de donnée initiaux correspondants dans les première et deuxième données comprimées pour générer des éléments de donnée intermédiaires correspondants.

5. Le processeur (109) selon la revendication 5, dans lequel le circuit, en réponse à l'instruction, multiplie simultanément dans un même cycle d'horloge les éléments de donnée initiaux correspondants dans les première et deuxième données comprimées pour générer des éléments de donnée intermédiaires correspondants.

6. Le processeur (109) selon la revendication 1, dans lequel le circuit (145) est un circuit de multiplication-addition comprenant un premier (813), deuxième (812), troisième (811) et quatrième (810) multiplicateur, chaque multiplicateur pour recevoir un ensemble correspondant des éléments de donnée ; le circuit de multiplication-addition comprenant en outre un premier additionneur (851) couplé au premier multiplicateur et au deuxième multiplicateur et un deuxième additionneur (850) couplé au troisième multiplicateur et au quatrième multiplicateur.

7. Le processeur (109) selon la revendication 6, dans lequel les deux éléments de donnée de résultat de la troisième donnée comprimée contiennent deux fois plus de bits que les éléments de donnée dans les première et deuxième données comprimées.

8. Le processeur (109) selon la revendication 7, dans lequel le circuit de multiplication-addition (145) comprend un premier Additionneur à Sauvegarde de Retenue (CSA) couplé aux premier (813) et deuxième (812) multiplicateur, le premier Additionneur à Sauvegarde de Retenue pour recevoir deux produits partiels générés par le premier multiplicateur (815) et recevoir deux produits partiels générés par le deuxième multiplicateur (812), le premier Additionneur à Sauvegarde de Retenue pour générer un premier ensemble de deux produits partiels, et un deuxième Additionneur à Sauvegarde de Retenue (CSA) couplé aux troisième (817) et quatrième (810) multiplicateur, le deuxième Additionneur à Sauvegarde de Retenue pour recevoir deux produits partiels générés par le troisième multiplicateur (811) et recevoir deux produits partiels générés par le quatrième multiplicateur (810), le deuxième Additionneur à Sauvegarde de Retenue pour générer un deuxième ensemble de deux produits partiels.

9. Le processeur (109) selon la revendication 8, dans lequel chaque multiplicateur exécute au moins une multiplication à 16 bits donnant pour résultat deux produits partiels.

10. Le processeur (109) selon la revendication 9, dans lequel les multiplicateurs sélectionnent chacun huit produits partiels et chaque multiplicateur comprend quatre niveaux d'Additionneur à Sauvegarde de Retenue.

11. Le processeur (109) selon la revendication 10, dans lequel chacun des quatre niveaux d'Additionneur à Sauve-

garde de Retenue compris dans chaque multiplicateur comprend un premier niveau réduisant les huit produits partiels à six produits partiels, un deuxième niveau réduisant les six produits partiels à quatre produits partiels, une troisième niveau réduisant les quatre produits partiels à trois produits partiels et un quatrième niveau réduisant les trois produits partiels à deux produits partiels.

12. Le processeur (109) selon la revendication 10, dans lequel le premier additionneur (851) est couplé au premier Additionneur à Sauvegarde de Retenue et le premier additionneur (851) génère une somme des deux produits partiels générés par le premier Additionneur à Sauvegarde de Retenue ; et le deuxième additionneur (850) est couplé au deuxième Additionneur à Sauvegarde de Retenue et le deuxième additionneur (850) génère une somme des deux produits partiels générés par le deuxième Additionneur à Sauvegarde de Retenue.

13. Le processeur (109) selon la revendication 8, dans lequel chaque multiplicateur réalise un encodage de Booth à 2 bits pour sélectionner un ensemble de huit produits partiels.

14. Le processeur (109) selon la revendication 13, dans lequel chaque multiplicateur réalise une méthode de génération de signe pour remplacer l'extension de signe des produits partiels.

15. Le processeur (109) selon la revendication 14, dans lequel chaque multiplicateur déplace un bit supplémentaire généré par la méthode de génération de signe à une position de bit vide dans une ligne séparée de produits partiels.

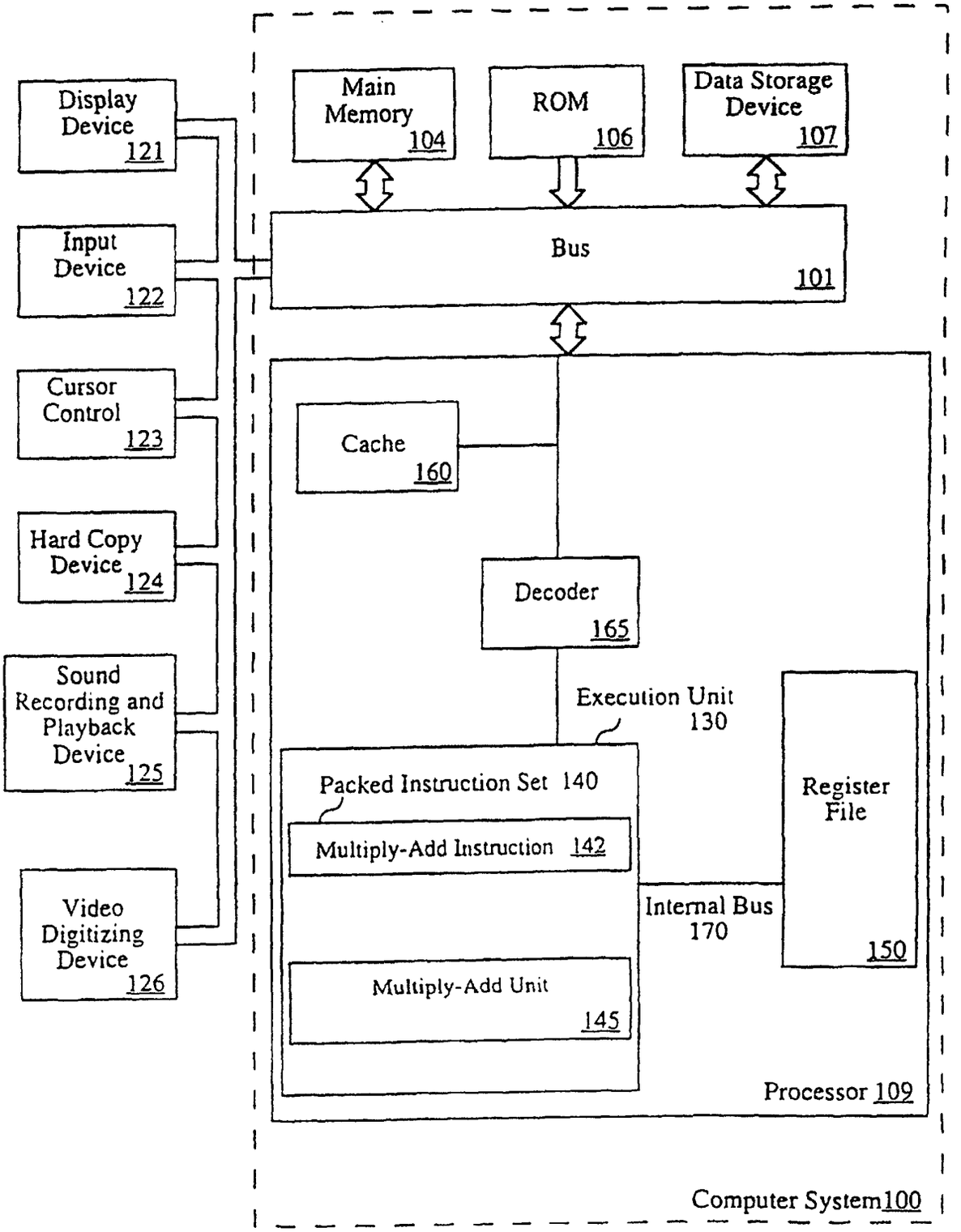

Display Device 121
Input Device 122
Cursor Control 123
Hard Copy Device 124
Sound Recording and Playback Device 125
Video Digitizing Device 126
Main Memory 104
ROM 106
Data Storage Device 107
Bus 101
Cache 160
Decoder 165
Execution Unit 130
Packed Instruction Set 140
Multiply-Add Instruction 142
Multiply-Add Unit 145
Internal Bus 170
Register File 150
Processor 109
Computer System 100

FIG. 1

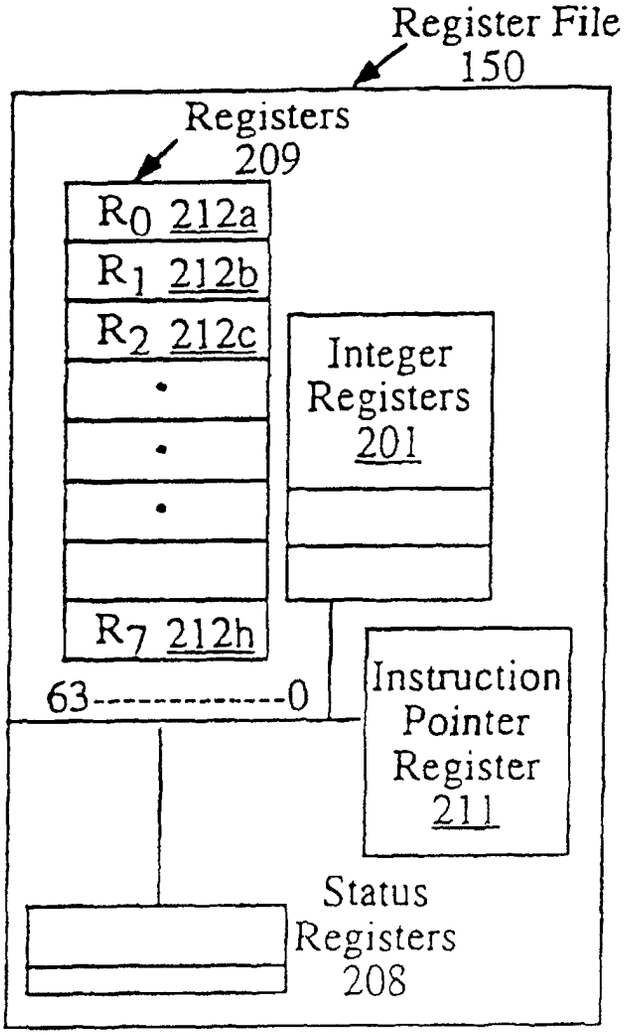

Register File
150
Registers
209
R0 212a
R1 212b
R2 212c
R7 212h
Integer
Registers
201
63----------0
Instruction
Pointer
Register
211
Status
Registers
208

FIG. 2

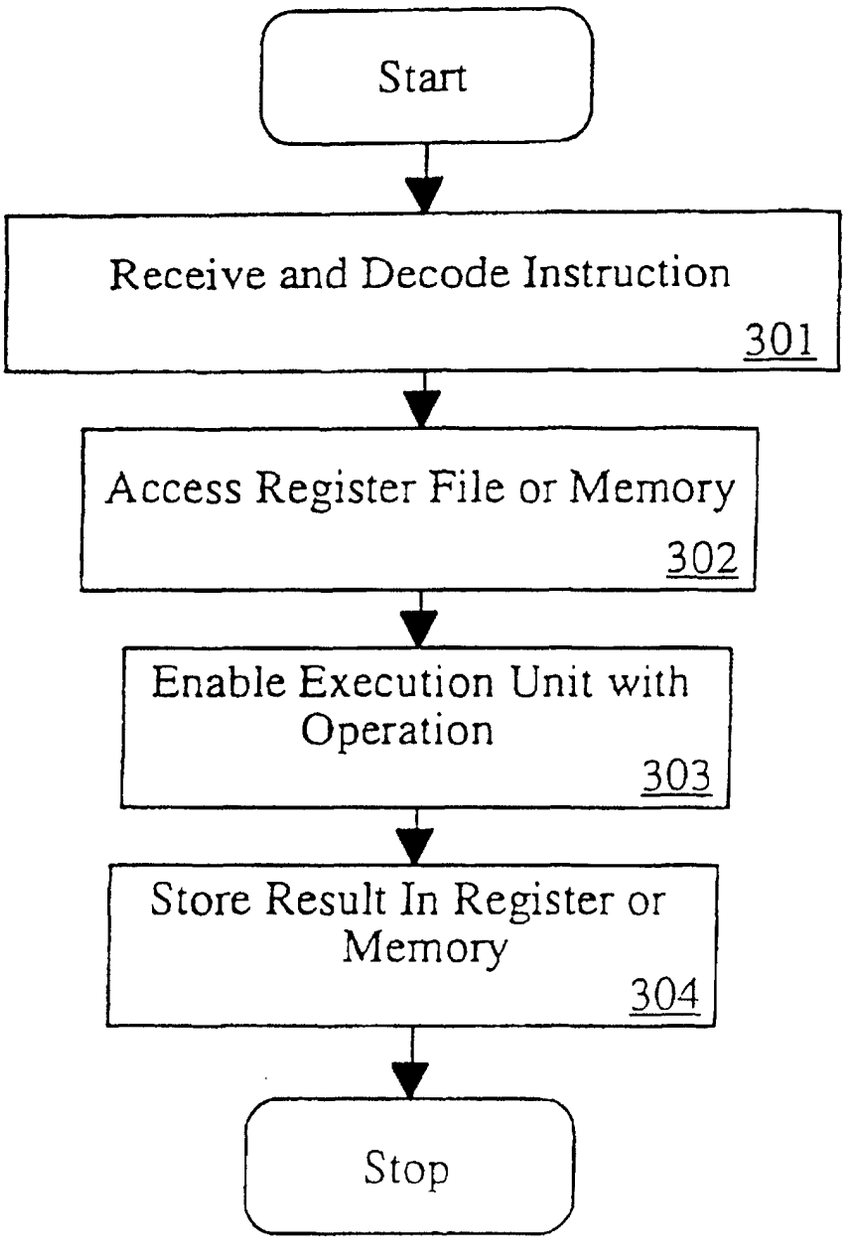

Start
Receive and Decode Instruction
301
Access Register File or Memory
302
Enable Execution Unit with Operation
303
Store Result In Register or Memory
304
Stop

FIG. 3

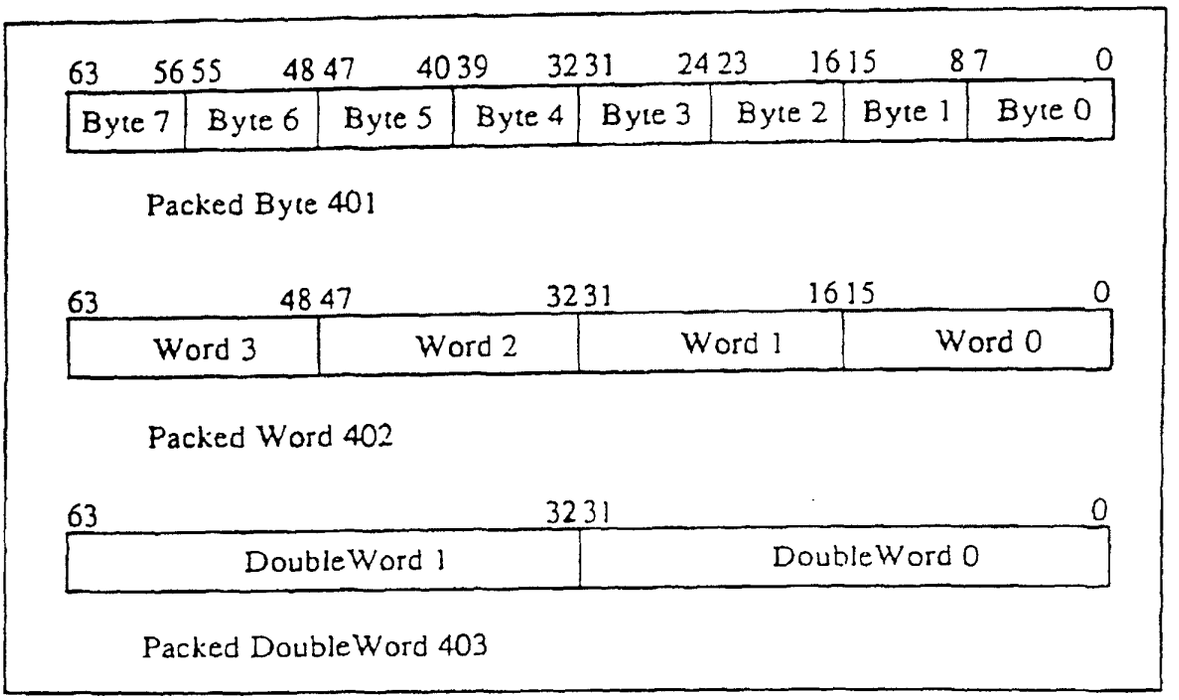

63 56 55 48 47 40 39 32 31 24 23 16 15 8 7 0
Byte 7
Byte 6
Byte 5
Byte 4
Byte 3
Byte 2
Byte 1
Byte 0
Packed Byte 401
63 48 47 32 31 16 15 0
Word 3
Word 2
Word 1
Word 0
Packed Word 402
63 32 31 0
DoubleWord 1
DoubleWord 0
Packed DoubleWord 403

FIG. 4

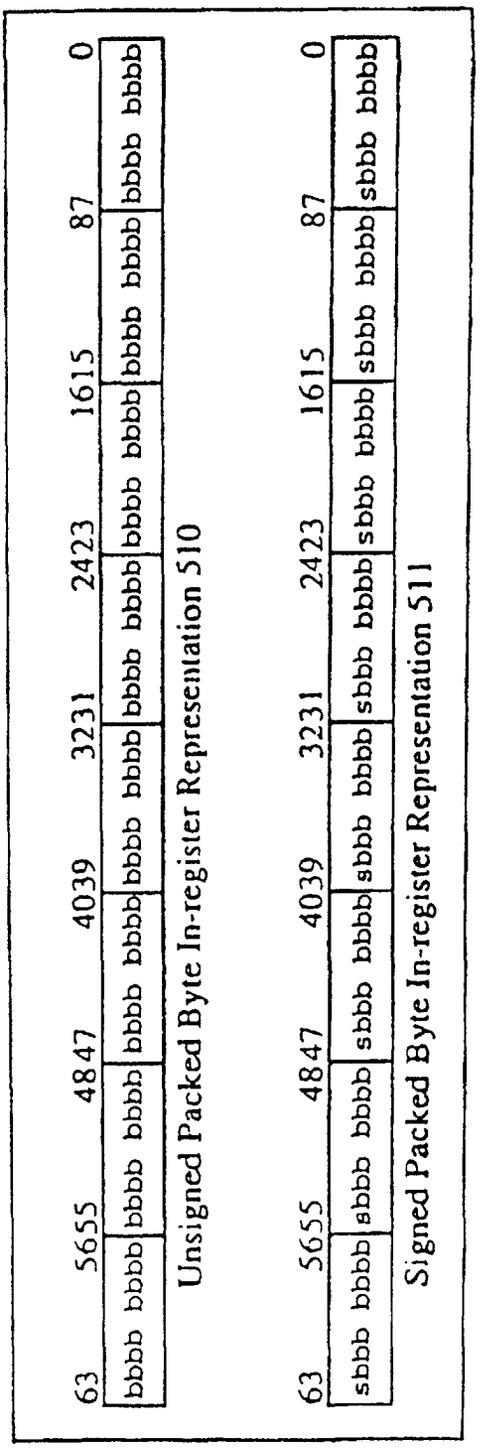

63
5655
4847
4039
3231
2423
1615
87
0
bbbb bbbb bbbb bbbb bbbb bbbb bbbb bbbb bbbb bbbb bbbb bbbb bbbb bbbb bbbb bbbb
Unsigned Packed Byte In-register Representation 510
63
5655
4847
4039
3231
2423
1615
87
0
sbbb bbbb sbbb bbbb sbbb bbbb sbbb bbbb sbbb bbbb sbbb bbbb sbbb bbbb sbbb bbbb
Signed Packed Byte In-register Representation 511

FIG. 5a

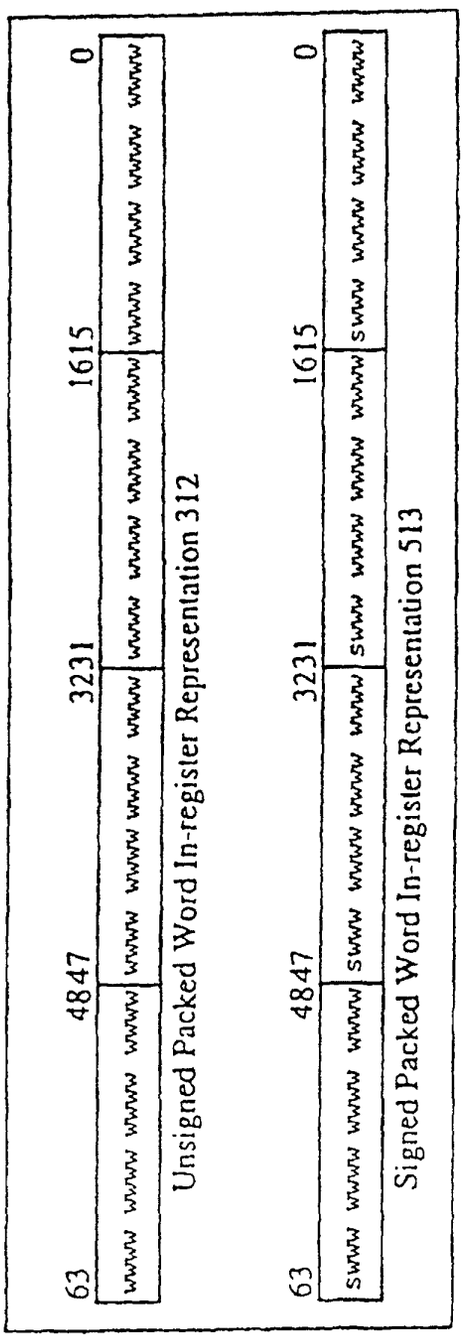

63
4847
3231
1615
0
wwww wwww wwww wwww
wwww wwww wwww wwww wwww wwww wwww wwww
Unsigned Packed Word In-register Representation 312
63
4847
3231
1615
0
swww wwww wwww wwww
swww wwww wwww wwww swww wwww wwww wwww
Signed Packed Word In-register Representation 513

FIG. 5b

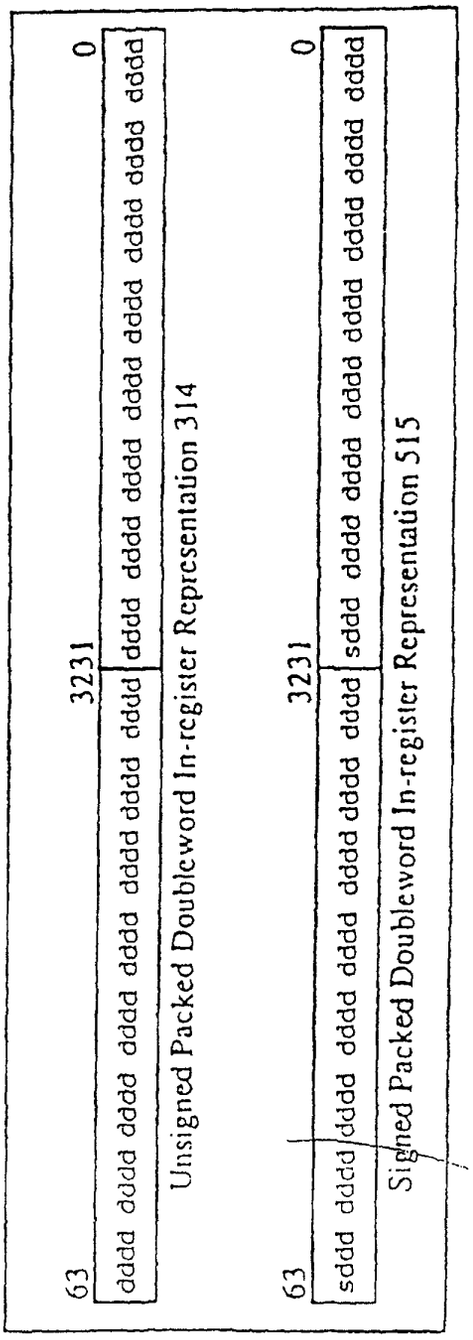

63
3231
0
dddd dddd dddd dddd
dddd dddd dddd dddd dddd dddd dddd dddd
Unsigned Packed Doubleword In-register Representation 314
63
3231
0
sddd dddd
sddd dddd dddd dddd
Signed Packed Doubleword In-register Representation 515

FIG. 5c

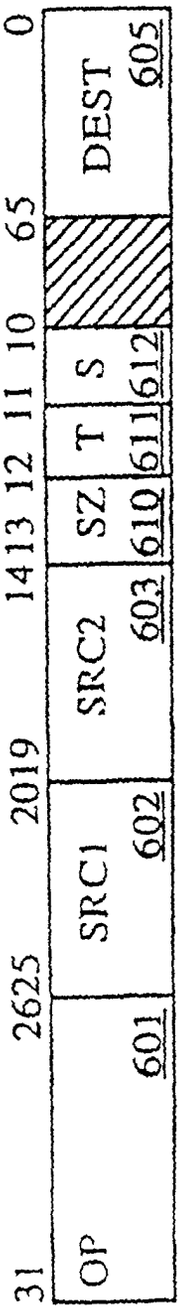

31
26 25
20 19
14 13 12 11 10
6 5
0
OP
601
SRC1
602
SRC2
603
SZ
610
T
611
S
612
DEST
605

FIG. 6a

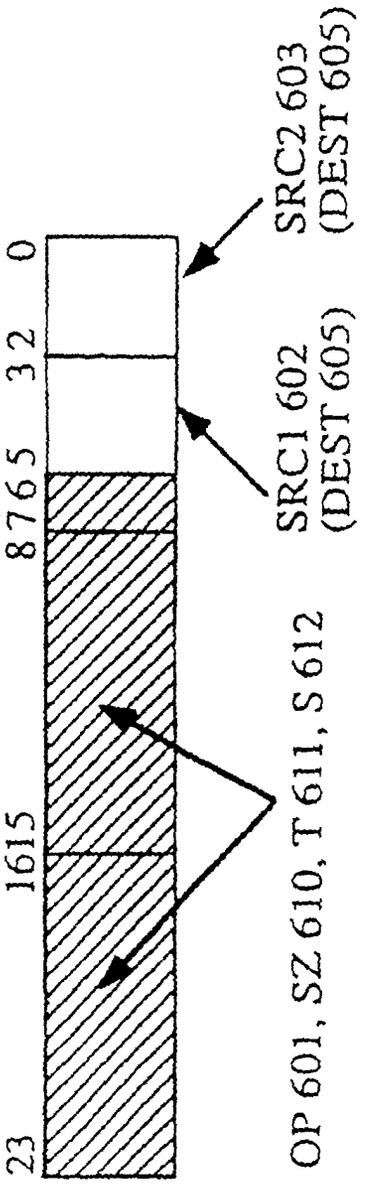

23
16 15
8 7 6 5
3 2
0
OP 601, SZ 610, T 611, S 612
SRC1 602
(DEST 605)
SRC2 603
(DEST 605)

FIG. 6b

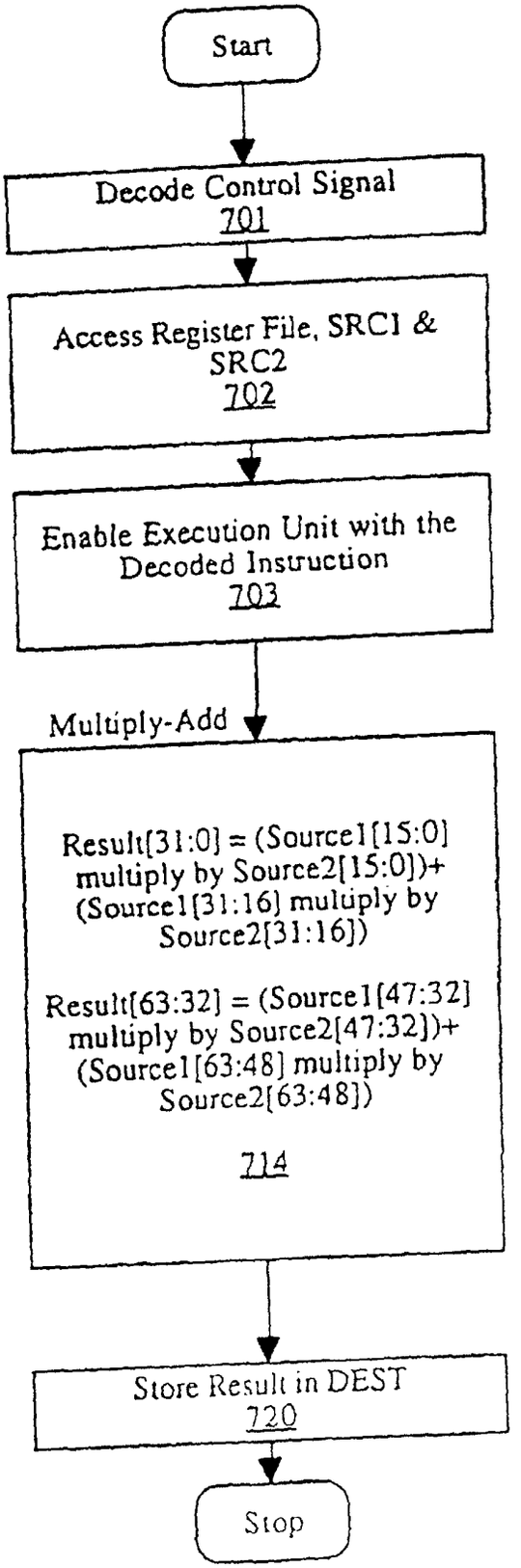

Start
Decode Control Signal
701
Access Register File, SRC1 &
SRC2
702
Enable Execution Unit with the
Decoded Instruction
703
Multiply-Add
Result[31:0] = (Source1[15:0]
multiply by Source2[15:0])+
(Source1[31:16] multiply by
Source2[31:16])
Result[63:32] = (Source1[47:32]
multiply by Source2[47:32])+
(Source1[63:48] multiply by
Source2[63:48])
714
Store Result in DEST
720
Stop

FIG. 7

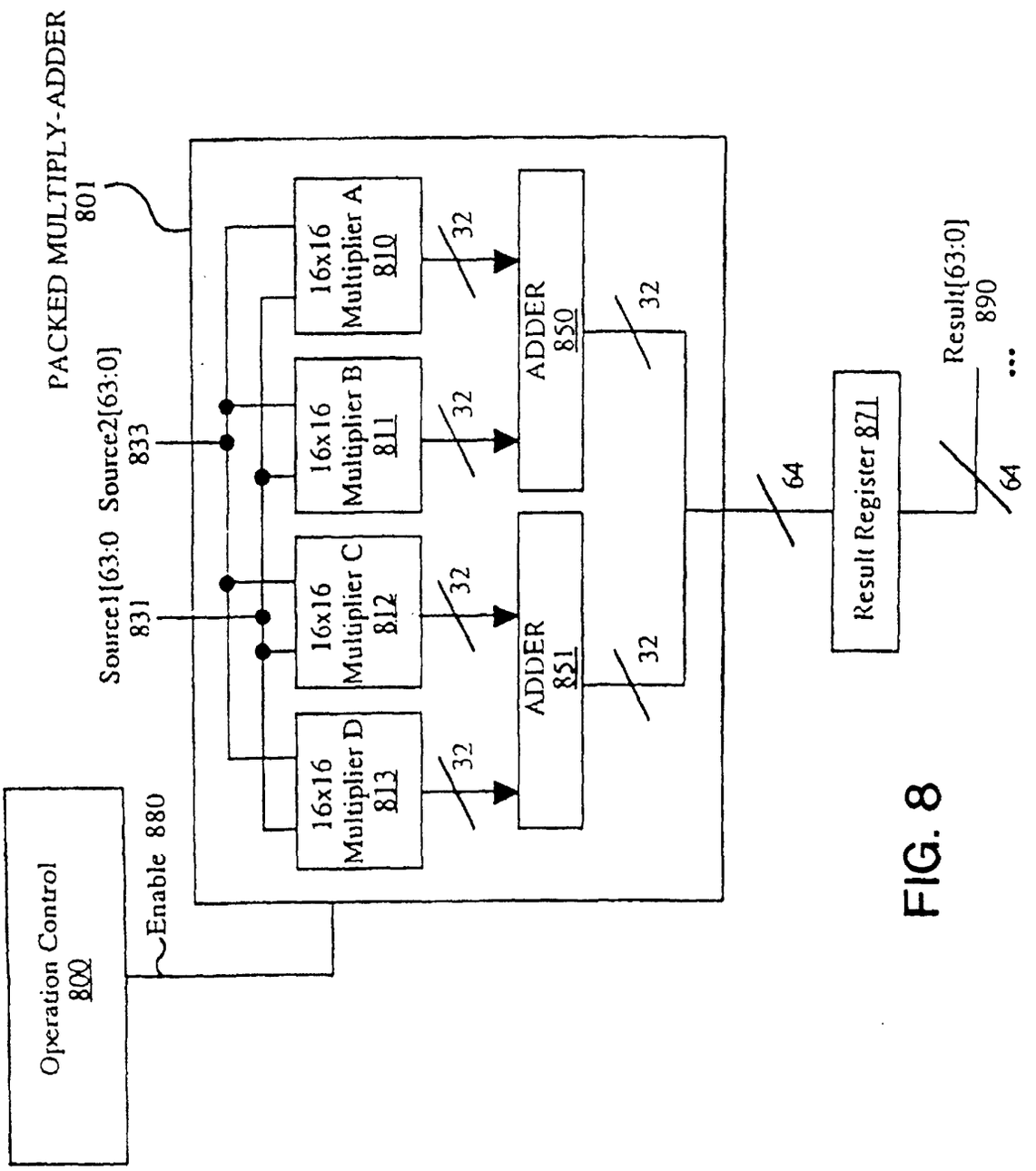

PACKED MULTIPLY-ADDER
801
Operation Control
800
Enable 880
Source1[63:0]
831
Source2[63:0]
833
16x16 Multiplier D
813
16x16 Multiplier C
812
16x16 Multiplier B
811
16x16 Multiplier A
810
32
ADDER
851
ADDER
850
64
Result Register 871
Result[63:0]
890
...

FIG. 8

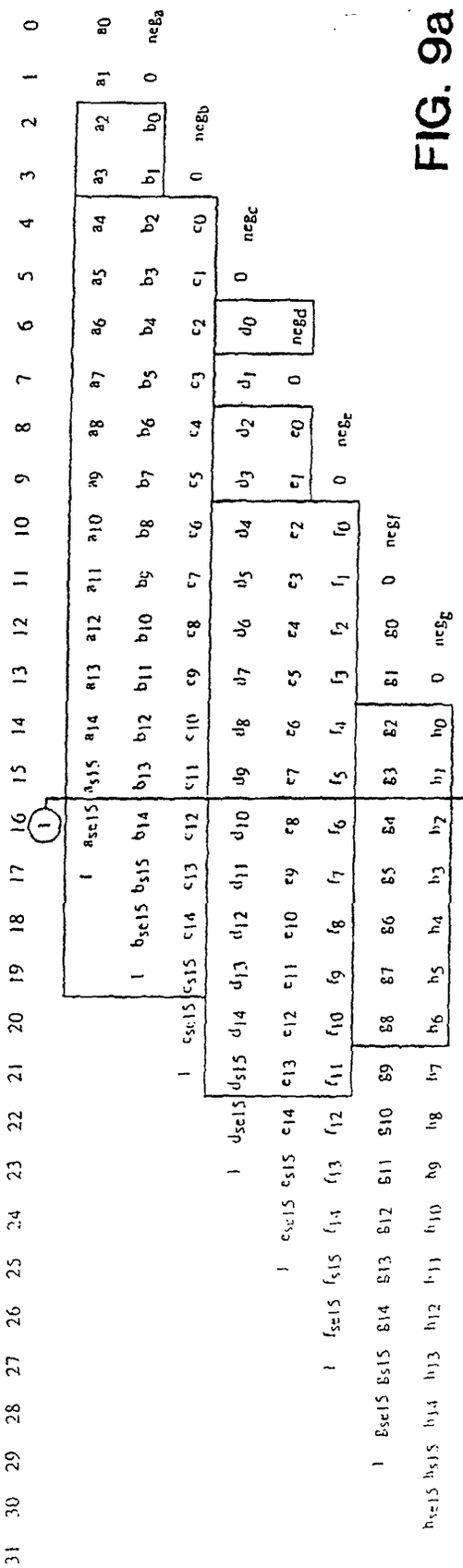


FIG. 9a

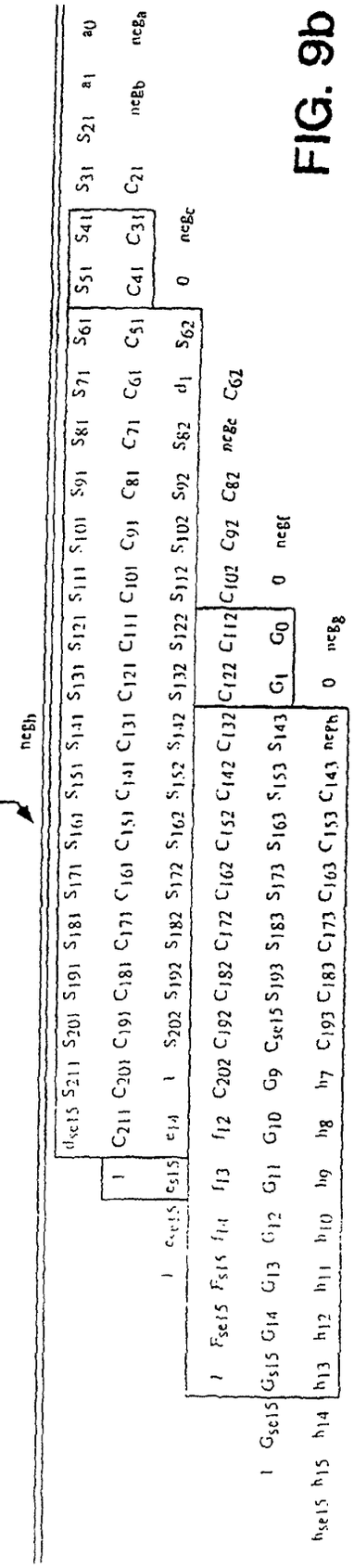


FIG. 9b

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				S_{271}	S_{261}	S_{251}	S_{241}	S_{231}	S_{221}	S_{212}	S_{203}	S_{194}	S_{184}	S_{174}	S_{164}	S_{154}	S_{144}	S_{133}	S_{123}	S_{113}	S_{103}	S_{93}	S_{83}	S_{72}	S_{63}	S_{52}	S_{42}	S_{31}	S_{21}	a_1	a_0
			C_{271}	C_{261}	C_{251}	C_{241}	C_{231}	C_{221}	C_{212}	C_{203}	C_{194}	C_{184}	C_{174}	C_{164}	C_{154}	C_{144}	C_{133}	C_{123}	C_{113}	C_{103}	C_{93}	C_{83}	C_{72}	C_{63}	C_{52}	C_{42}	neg_c	C_{21}	neg_b	0	neg_a
		1	G_{se15}	0	0	1	e_{sc5}	S_{232}	S_{222}	S_{213}	S_{204}	S_{195}	S_{185}	S_{175}	S_{165}	S_{155}	S_{145}	S_{134}	S_{124}	C_{102}	C_{92}	C_{82}	neg_e	C_{62}								
1	h_{se15}	h_{s15}	h_{14}	0	0	0	C_{232}	C_{222}	C_{213}	C_{204}	C_{195}	C_{185}	C_{175}	C_{165}	C_{155}	C_{145}	C_{134}	C_{124}	neg_g	0	neg_f											

FIG. 9c

		G_{sc15}	S_{271}	S_{261}	S_{252}	S_{242}	S_{233}	S_{223}	S_{214}	S_{205}	S_{196}	S_{186}	S_{176}	S_{166}	S_{156}	S_{146}	S_{135}	S_{125}	S_{114}	S_{104}	S_{94}	S_{84}	S_{73}	S_{63}	S_{52}	S_{42}	S_{31}	S_{21}	a_1	a_0
	1	C_{271}	C_{261}	C_{252}	C_{242}	C_{233}	C_{223}	C_{214}	C_{205}	C_{196}	C_{186}	C_{176}	C_{166}	C_{156}	C_{146}	C_{135}	C_{125}	C_{114}	C_{104}	C_{94}	C_{84}	C_{73}	C_{63}	C_{52}	C_{42}	neg_c	C_{21}	neg_b	0	neg_a
1	h_{sc15}	h_{s15}	h_{14}	0	C_{251}	0	C_{232}	C_{222}	C_{213}	C_{204}	C_{195}	C_{185}	C_{175}	C_{165}	C_{155}	C_{145}	C_{134}	C_{124}	neg_g	0	neg_f											

FIG. 9d

1	h_{se15}	C_{291}	C_{281}	S_{272}	S_{262}	S_{253}	S_{243}	S_{234}	S_{224}	S_{215}	S_{206}	S_{197}	S_{187}	S_{177}	S_{167}	S_{157}	S_{147}	S_{136}	S_{126}	S_{115}	S_{105}	S_{94}	S_{84}	S_{73}	S_{63}	S_{52}	S_{42}	S_{31}	S_{21}	a_1	a_0
0	C_{291}	C_{281}	C_{272}	C_{262}	C_{253}	C_{243}	C_{234}	C_{224}	C_{215}	C_{206}	C_{197}	C_{187}	C_{177}	C_{167}	C_{157}	C_{147}	C_{136}	C_{126}	C_{115}	C_{105}	neg_f	C_{84}	C_{73}	C_{62}	C_{52}	C_{42}	neg_c	C_{21}	neg_b	0	neg_a

FIG. 9e

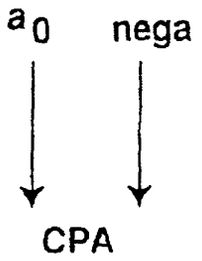


FIG. 10A

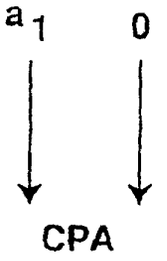


FIG. 10B

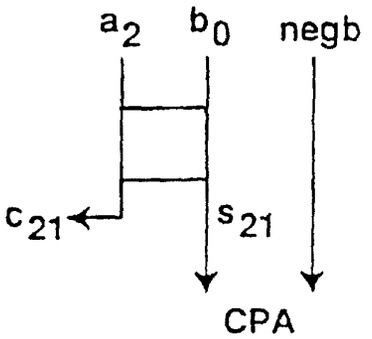


FIG. 10C

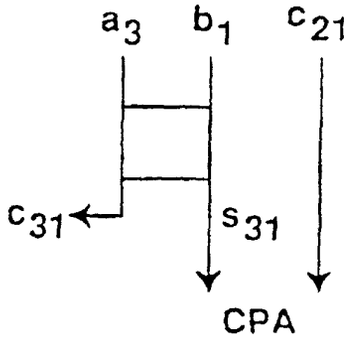


FIG. 10D

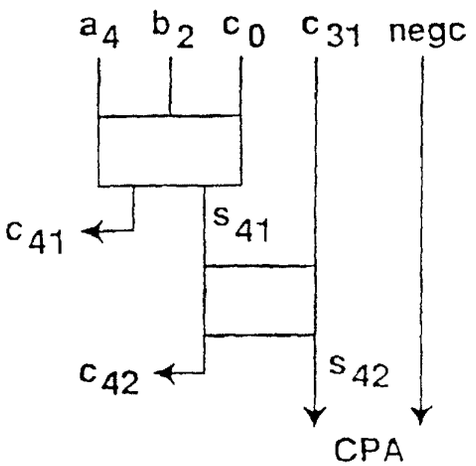


FIG. 10E

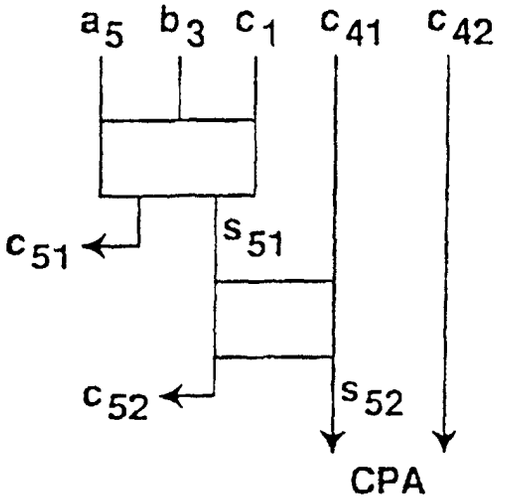


FIG. 10F

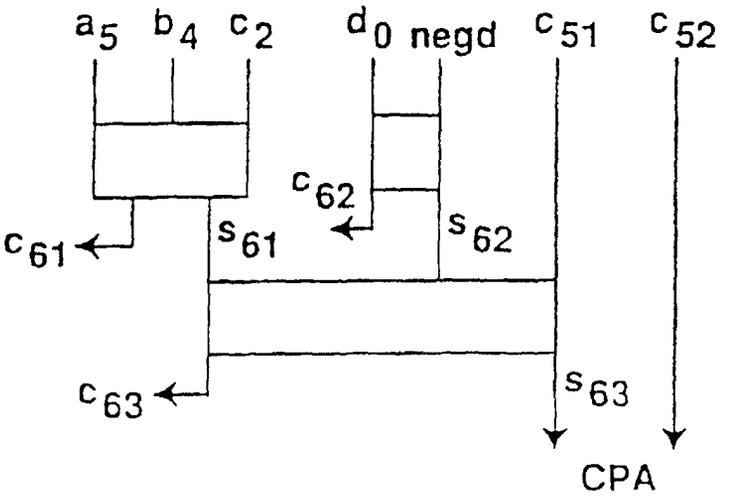


FIG. 10G

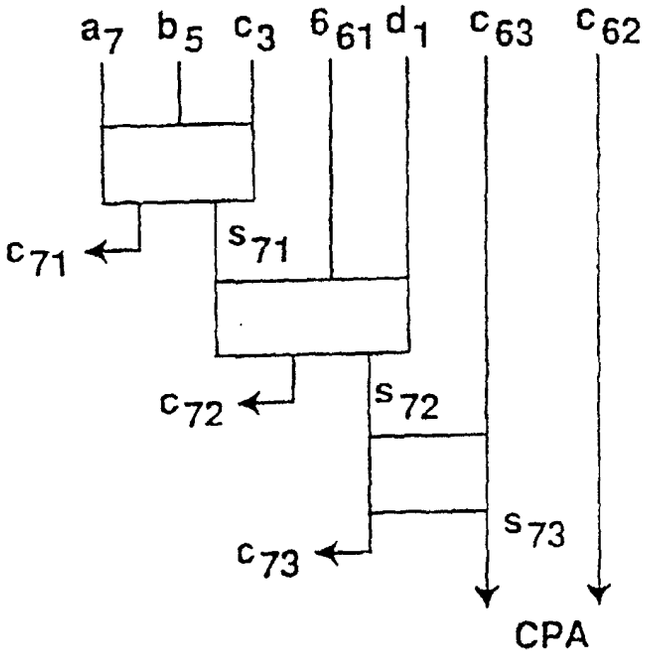


FIG. 10H

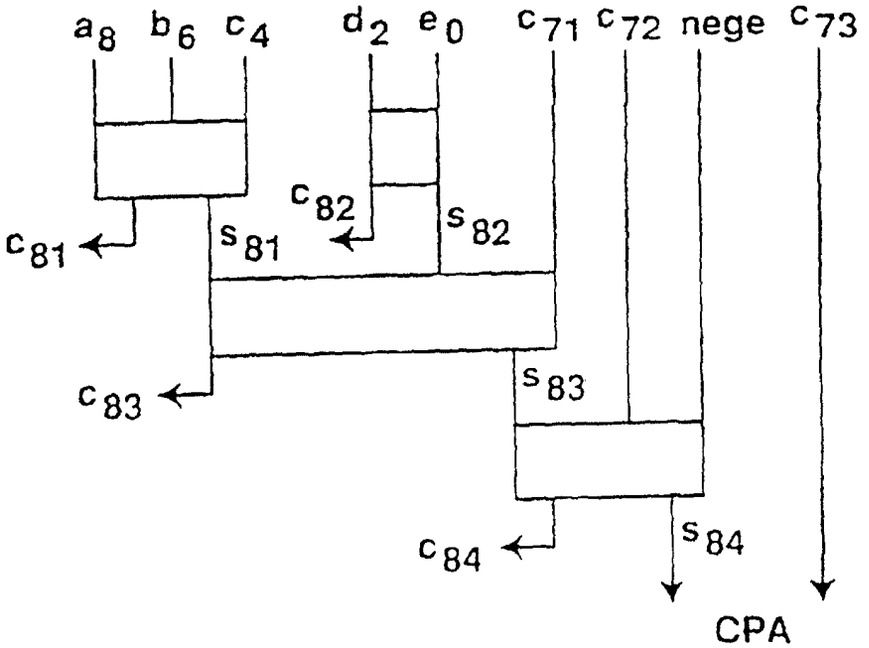


FIG. 10I

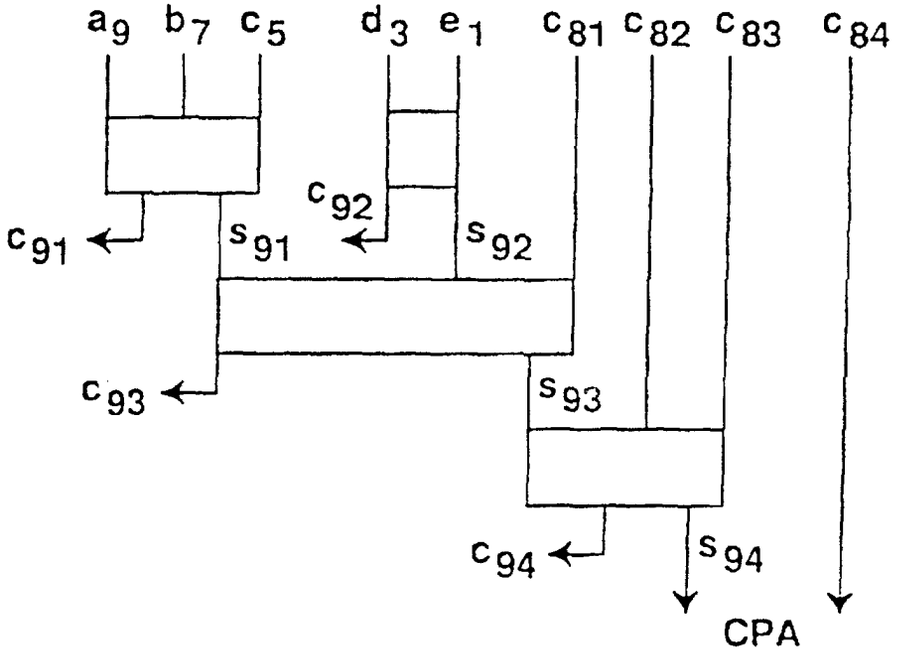


FIG. 10J

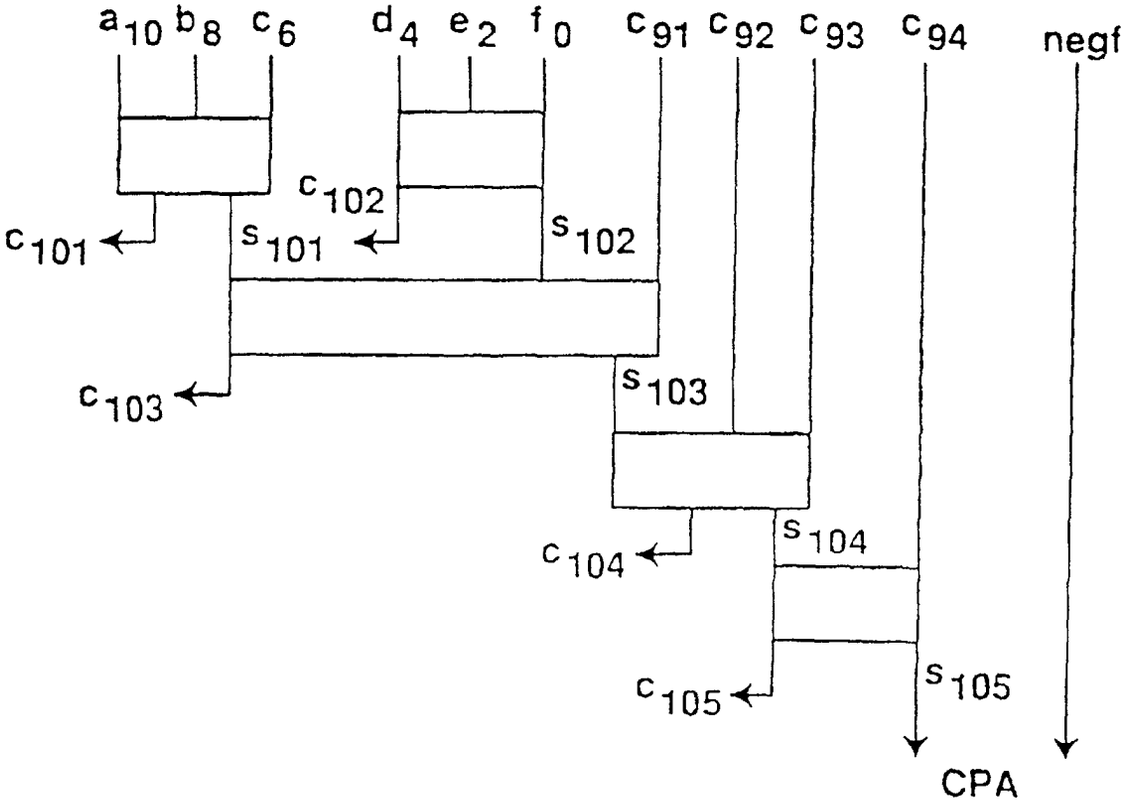


FIG. 10K

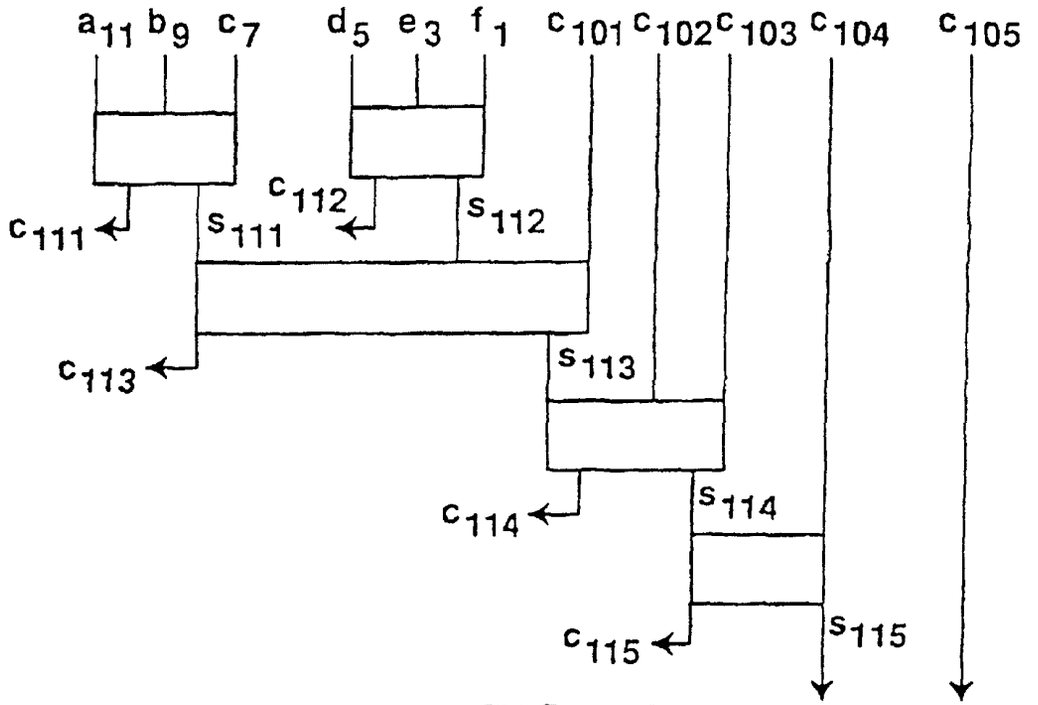


FIG. 10L

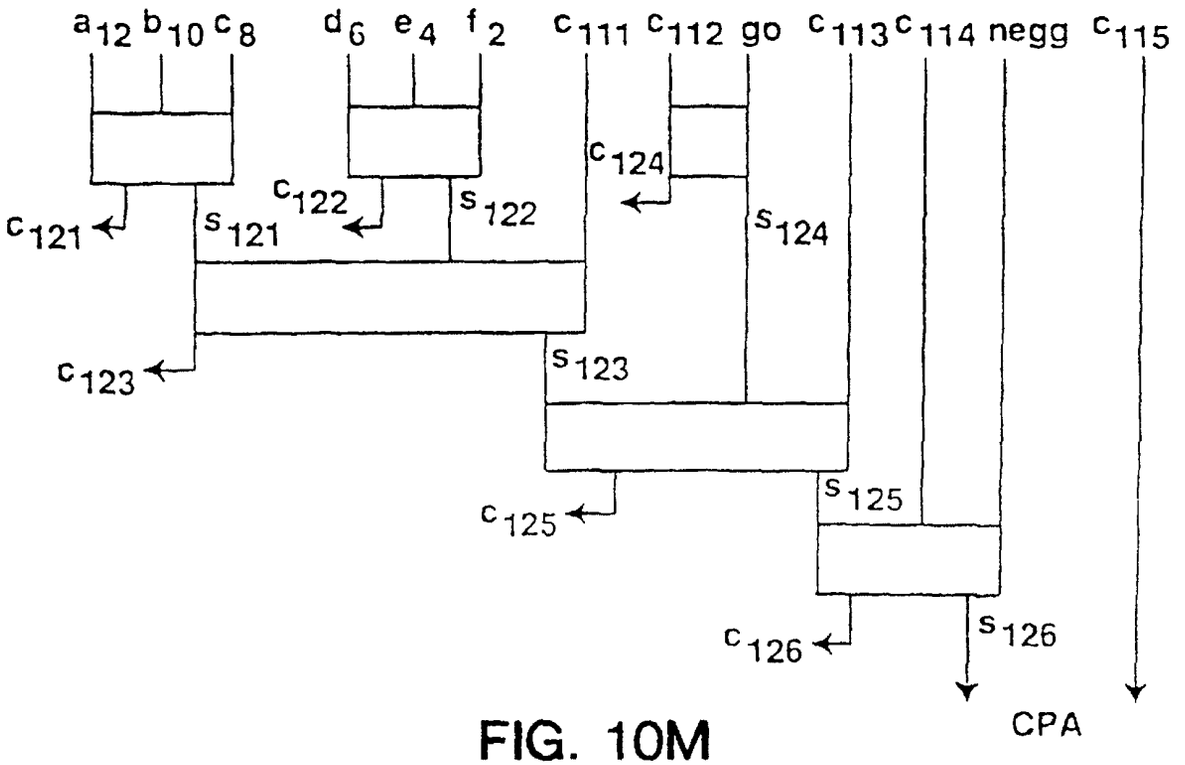


FIG. 10M

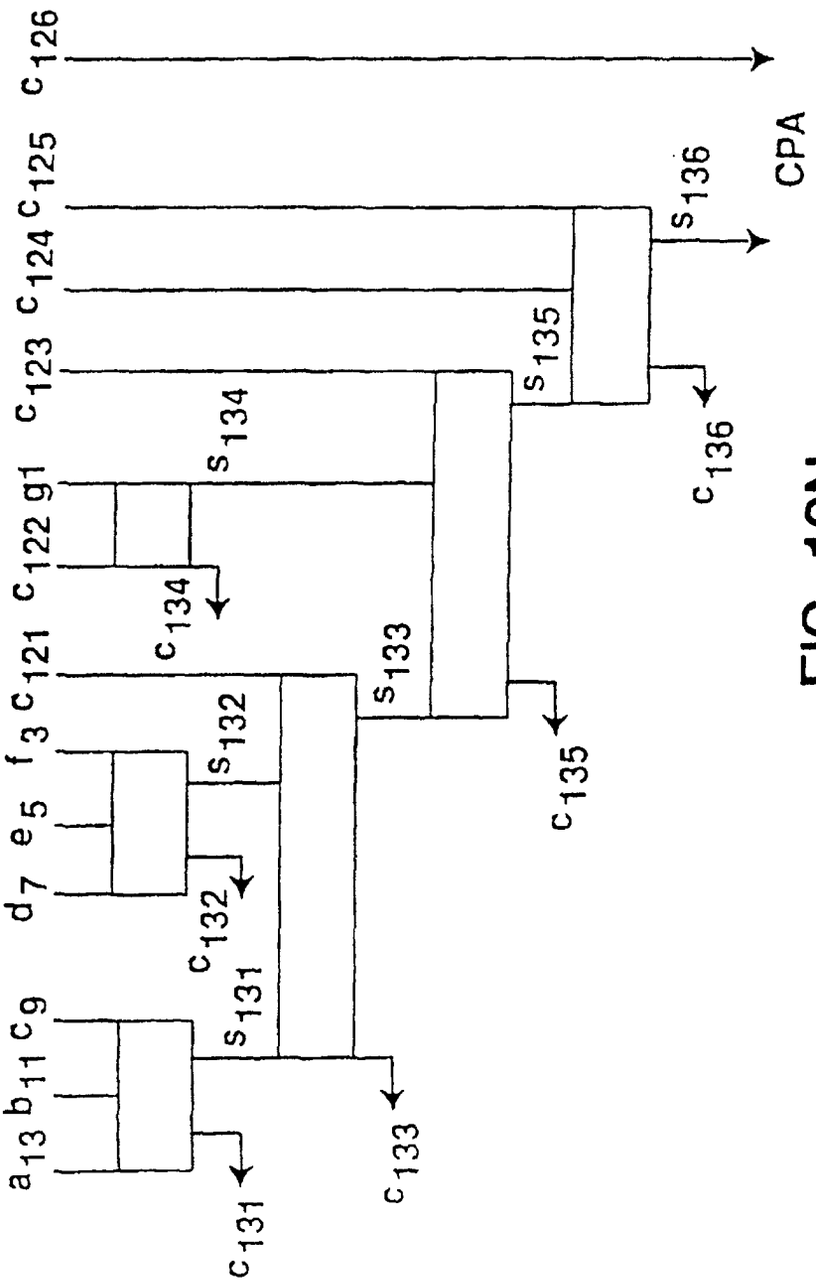

c126
c125
c124
c123
c122
c121
f3
e5
d7
g1
c9
b11
a13
CPA
s136
s135
s134
s133
s132
s131
c136
c135
c134
c132
c131
c133

FIG. 10N

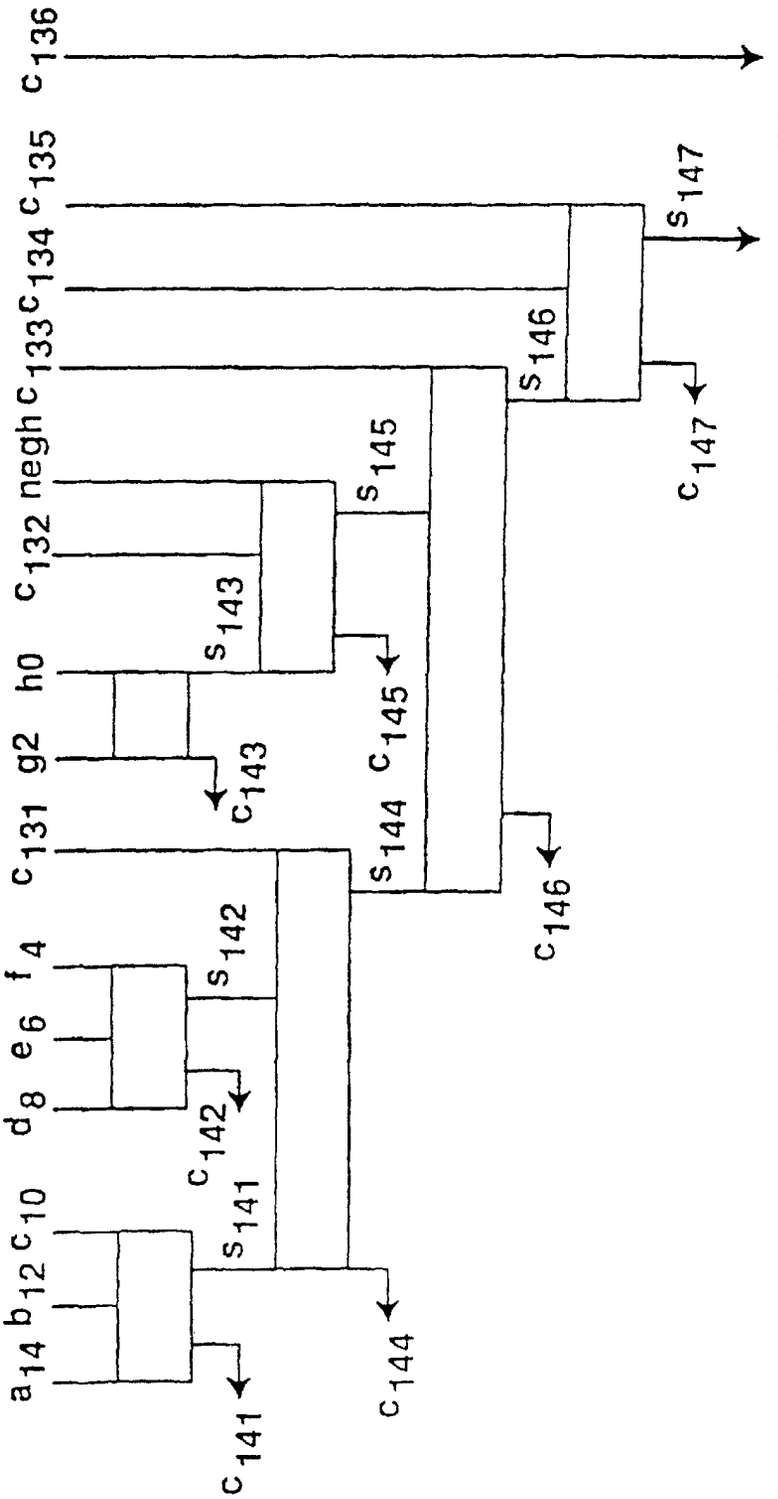

c136
c135
c134
c133
negh
c132
h0
g2
c131
f4
e6
d8
c10
b12
a14
CPA
s147
s146
s145
s144
s143
s142
s141
c147
c146
c145
c144
c143
c142
c141

FIG. 100

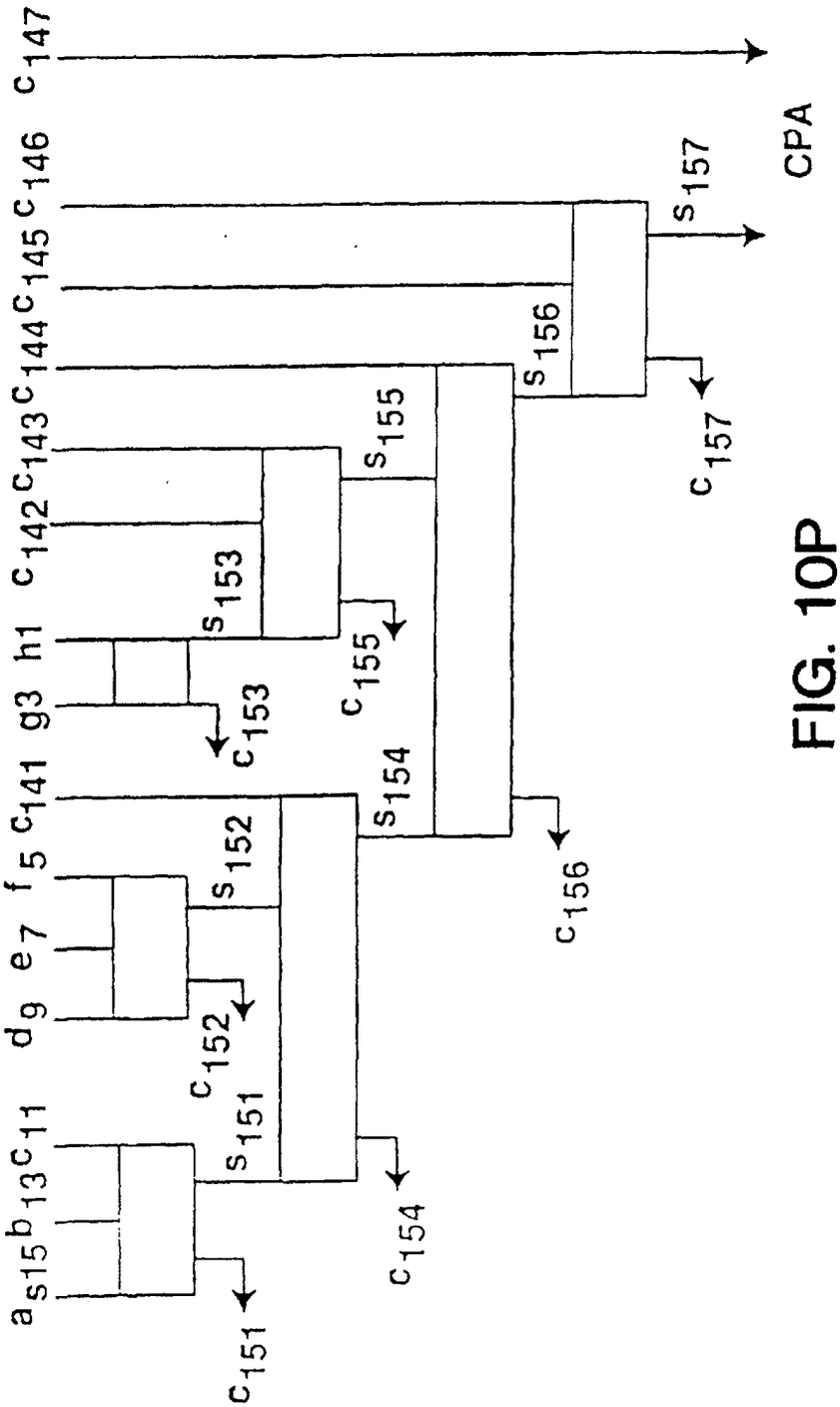

c147
CPA
s157
c146
c145
c144
s156
c157
c143
s155
c142
s153
h1
c155
g3
c153
c141
s154
f5
s152
e7
c156
d9
c152
s151
c11
b13
c154
as15
c151

FIG. 10P

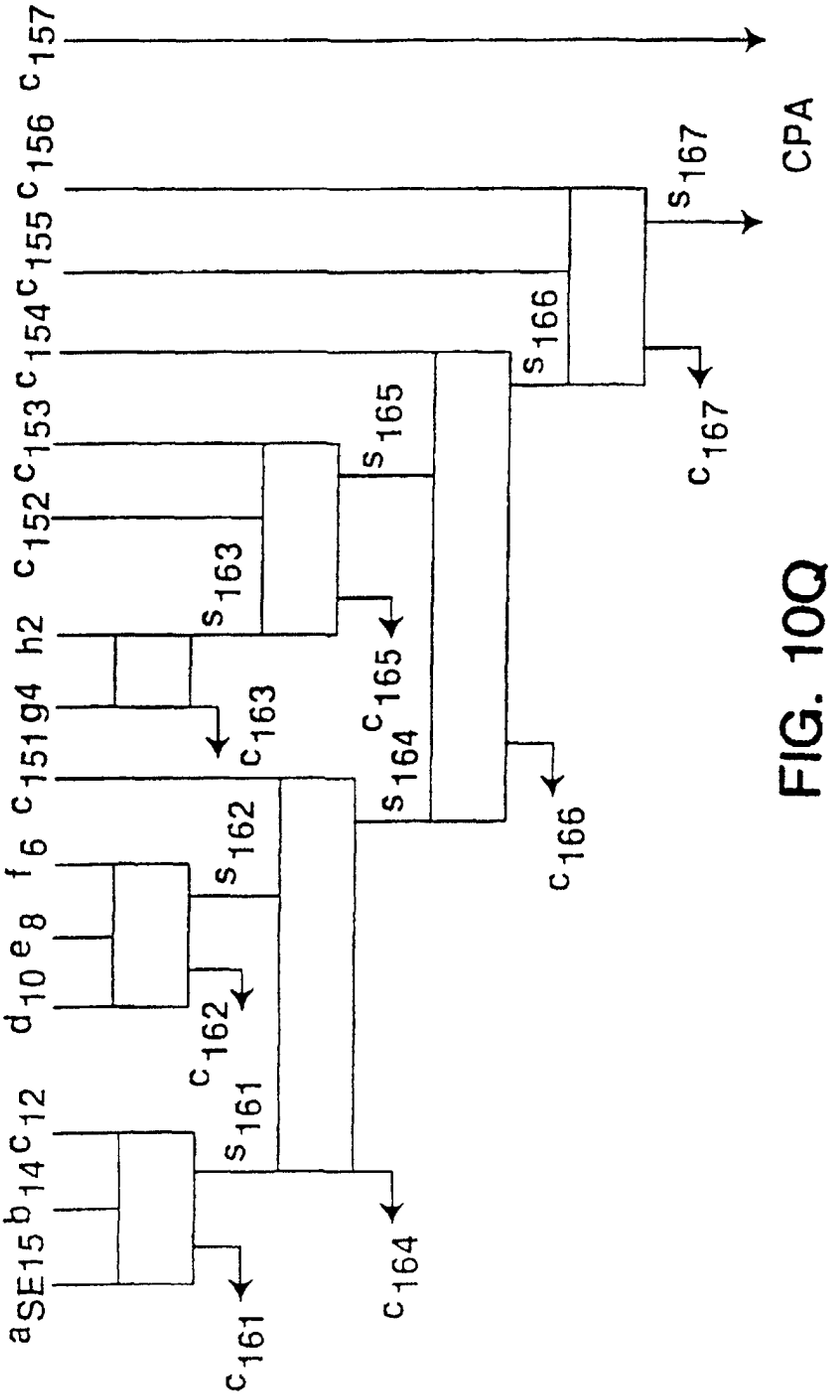

a SE15
b 14
c 12
d 10
e 8
f 6
g 4
h 2
c 151
c 152
c 153
c 154
c 155
c 156
c 157
s 161
s 162
s 163
s 164
s 165
s 166
s 167
c 161
c 162
c 163
c 164
c 165
c 166
c 167
CPA

FIG. 10Q

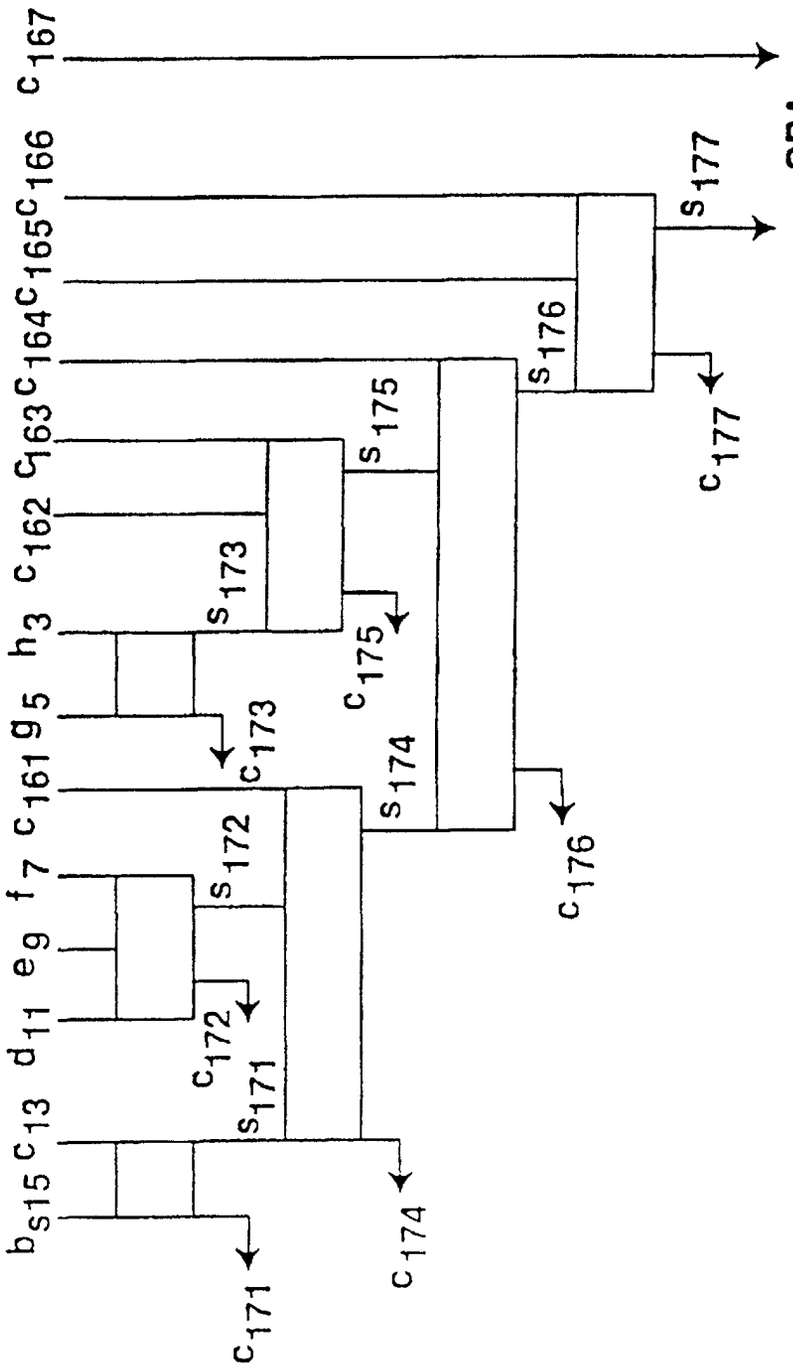

c167
c166
c165
c164
c163
c162
h3
g5
c161
f7
e9
d11
c13
bs15
CPA
s177
s176
s175
s174
s173
s172
s171
c177
c176
c175
c174
c173
c172
c171

FIG. 10R

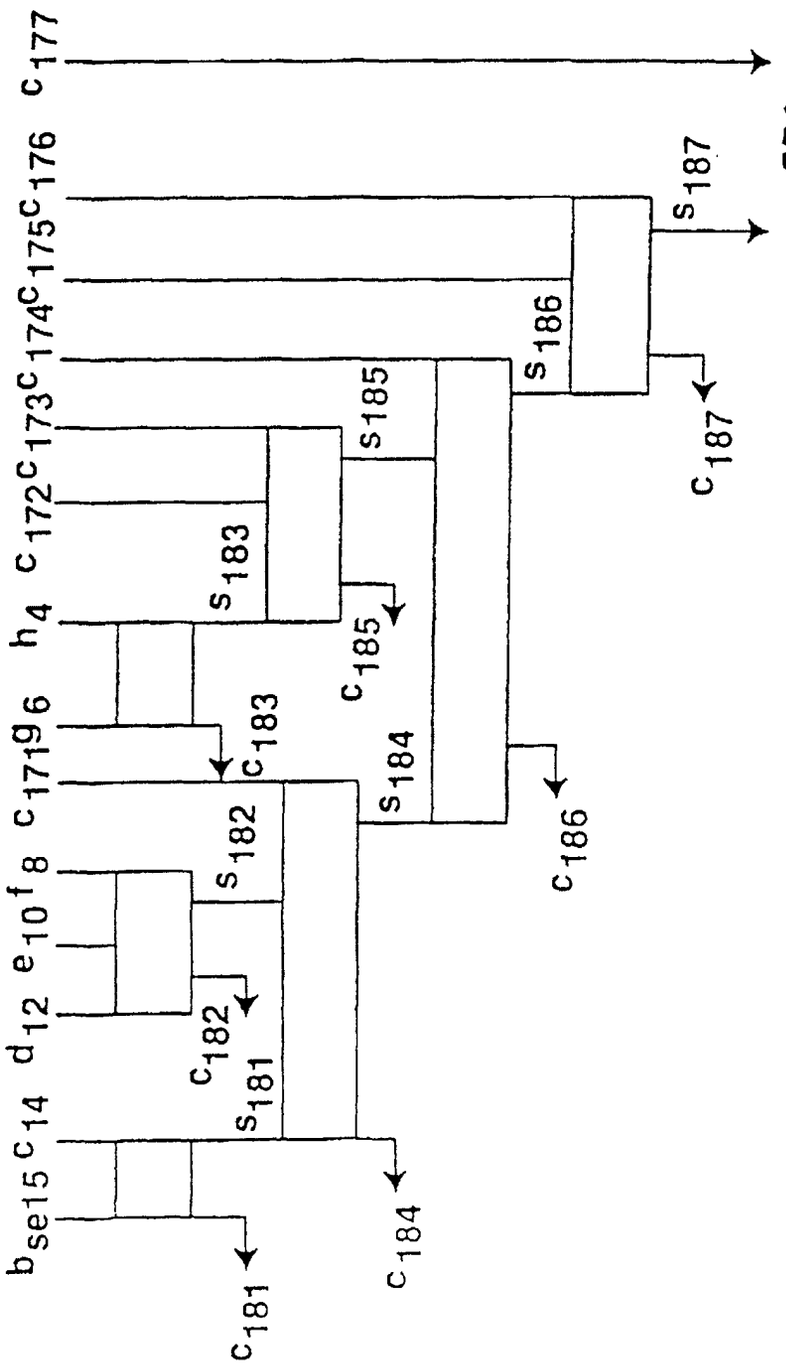

c_{177}
c_{176}
c_{175}
c_{174}
c_{173}
c_{172}
h_4
g_6
c_{171}
f_8
e_{10}
d_{12}
c_{14}
b_{se15}
CPA
s_{187}
s_{186}
s_{185}
s_{184}
s_{183}
s_{182}
s_{181}
c_{187}
c_{186}
c_{185}
c_{184}
c_{183}
c_{182}
c_{181}

FIG. 10S

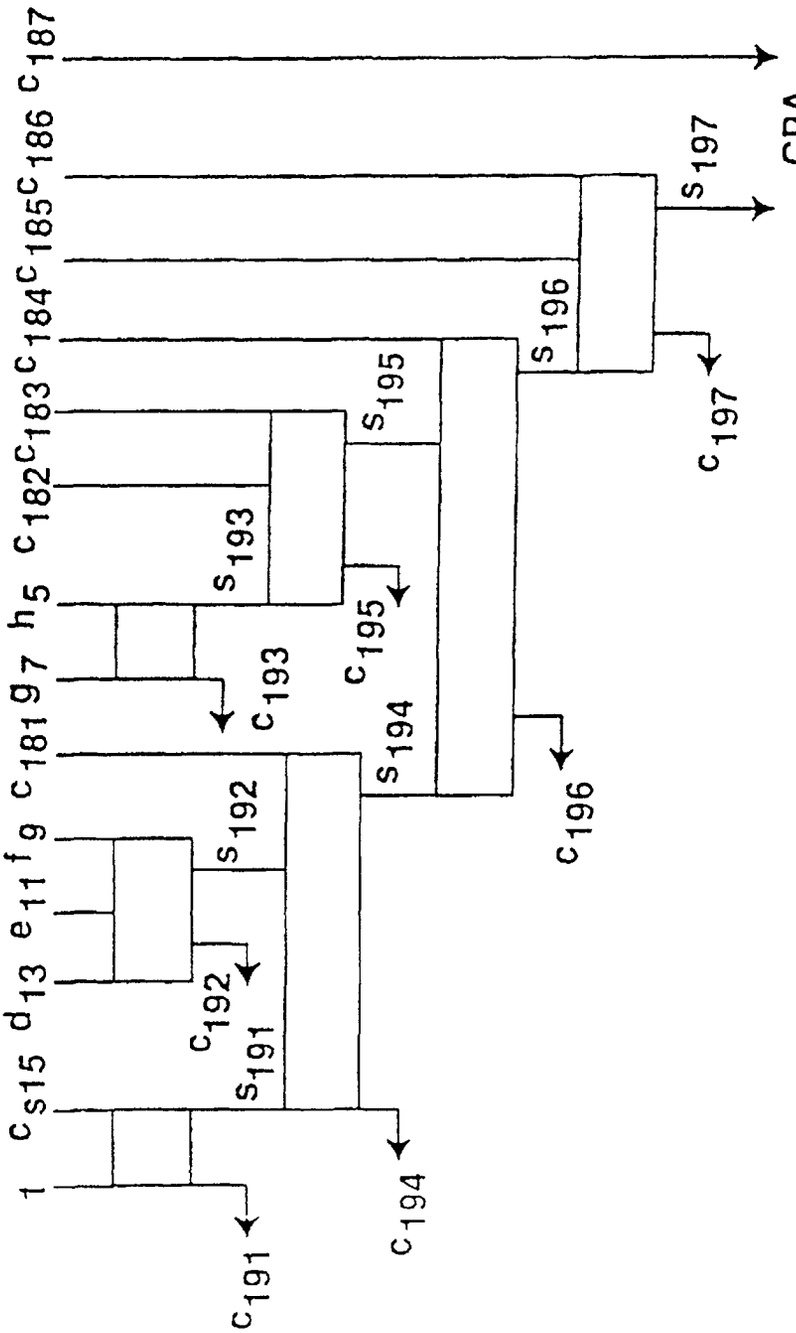

1
c_{s15}
d_{13}
e_{11}
f_9
c_{181}
g_7
h_5
c_{182}
c_{183}
c_{184}
c_{185}
c_{186}
c_{187}
c_{191}
c_{192}
s_{191}
s_{192}
c_{193}
s_{193}
c_{194}
c_{195}
s_{194}
s_{195}
c_{196}
s_{196}
c_{197}
s_{197}
CPA

FIG. 10T

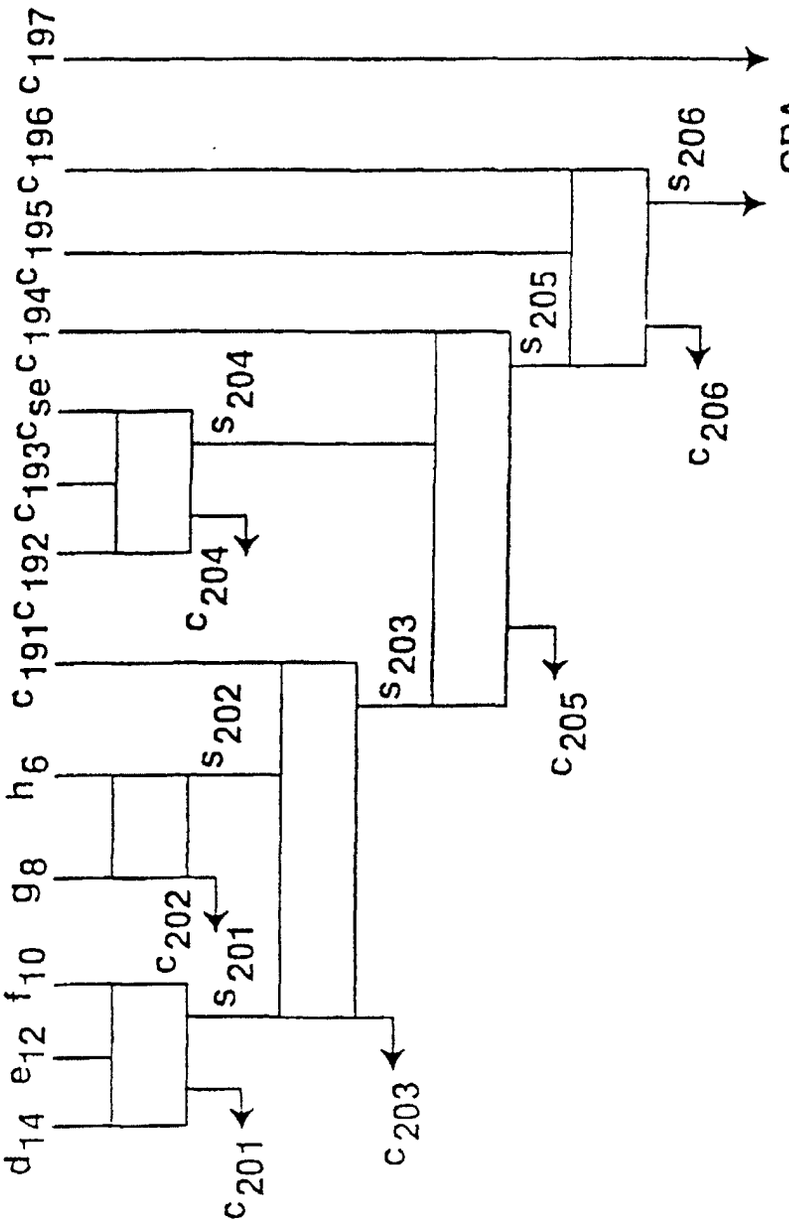

d14
e12
f10
g8
h6
c191
c192
c193
cse
c194
c195
c196
c197
c201
c202
c203
c204
c205
c206
s201
s202
s203
s204
s205
s206
CPA

FIG. 10U

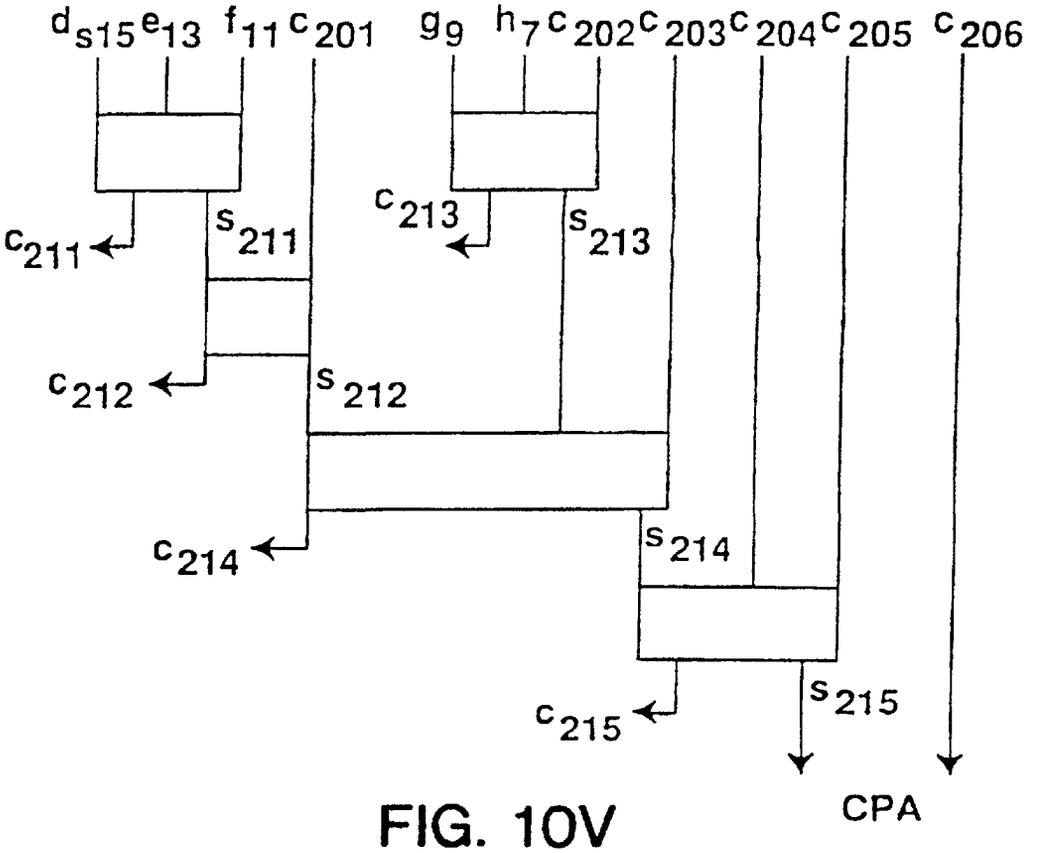


FIG. 10V

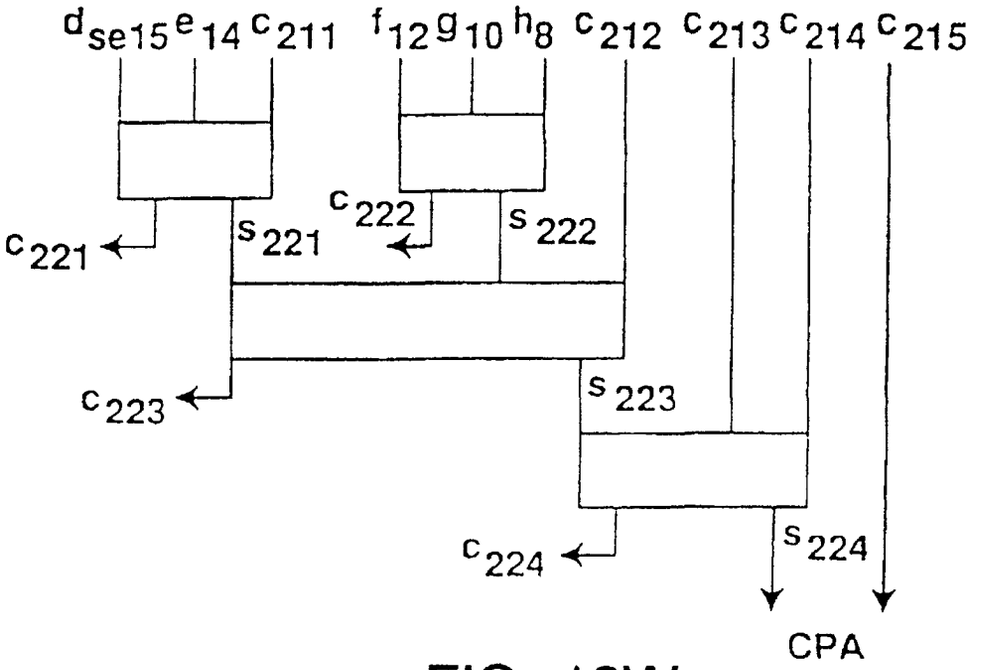


FIG. 10W

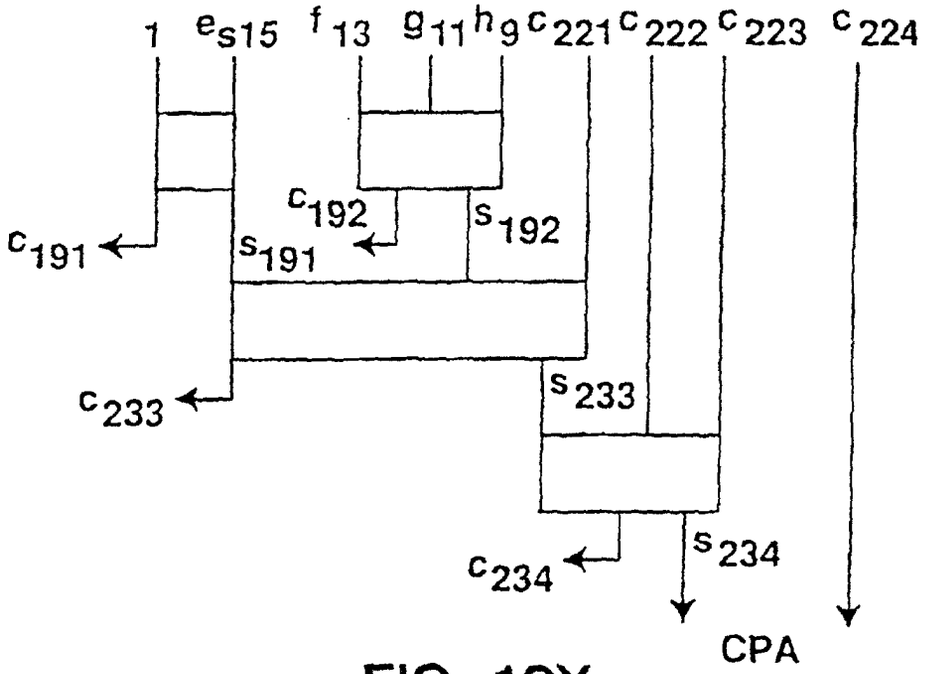


FIG. 10X

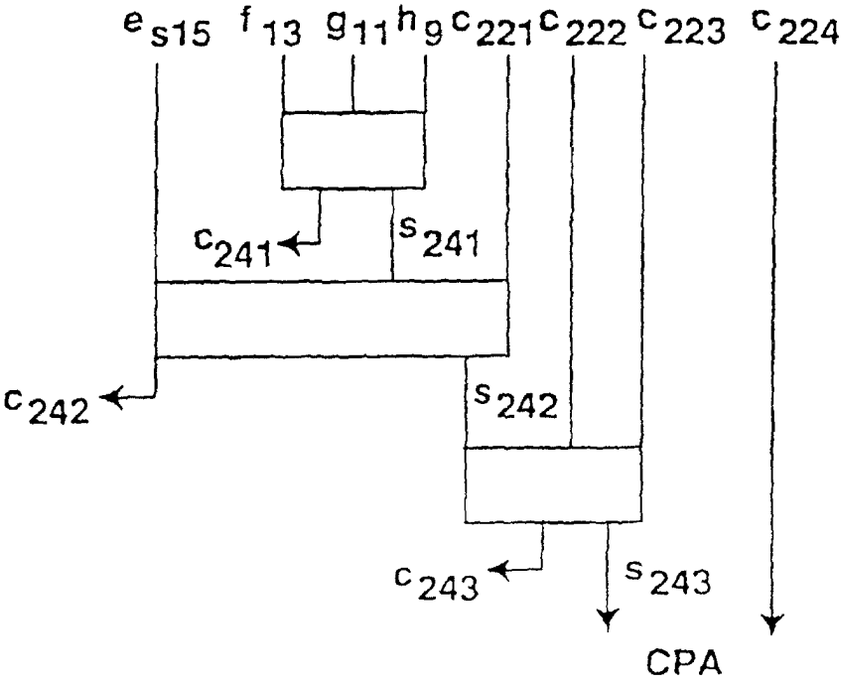


FIG. 10Y

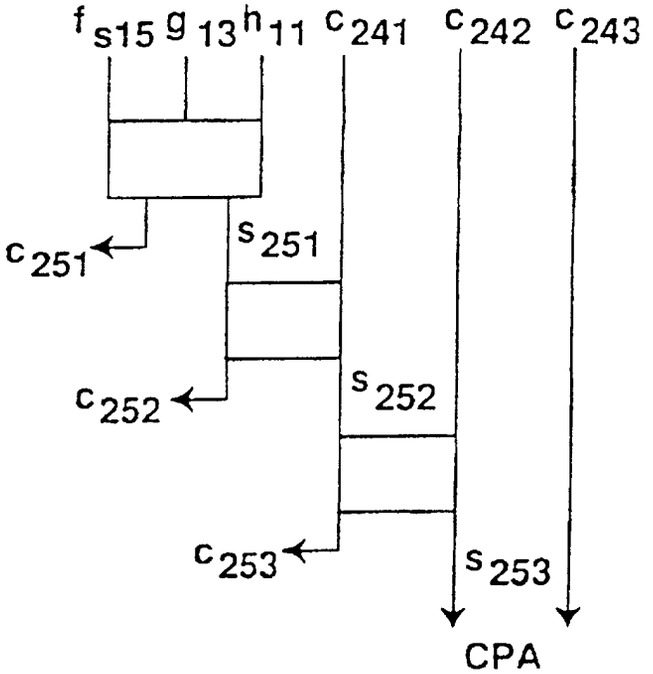


FIG. 10Z

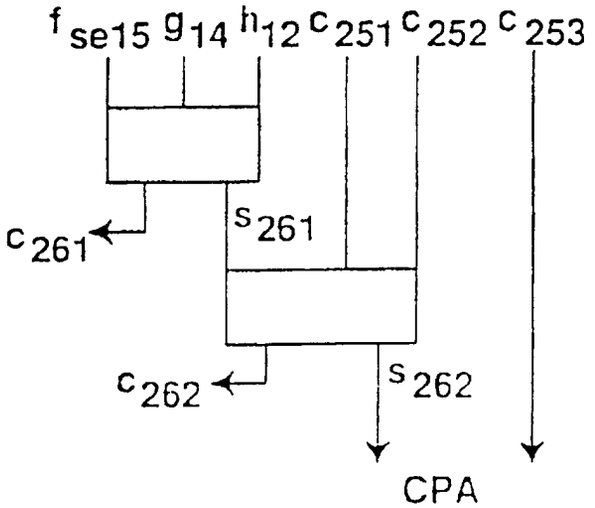


FIG. 10AA

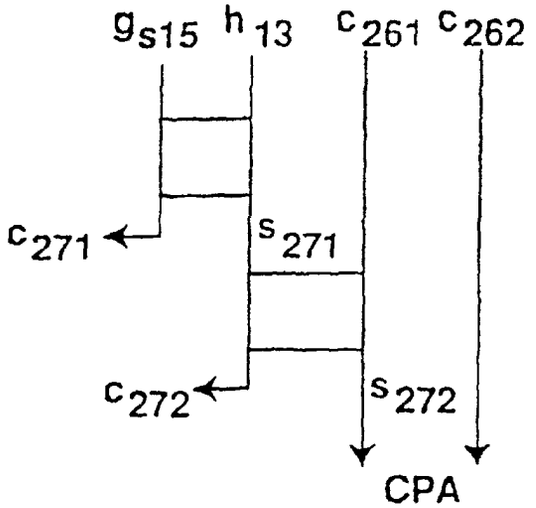


FIG. 10AB

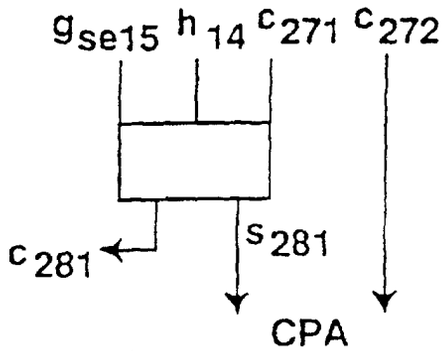


FIG. 10AC

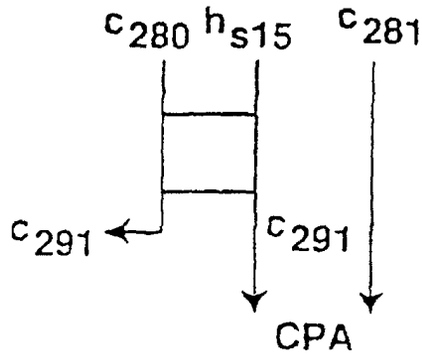


FIG. 10AD

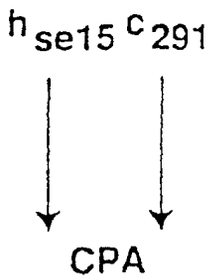


FIG. 10AE

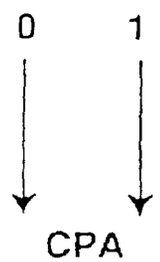


FIG. 10AF

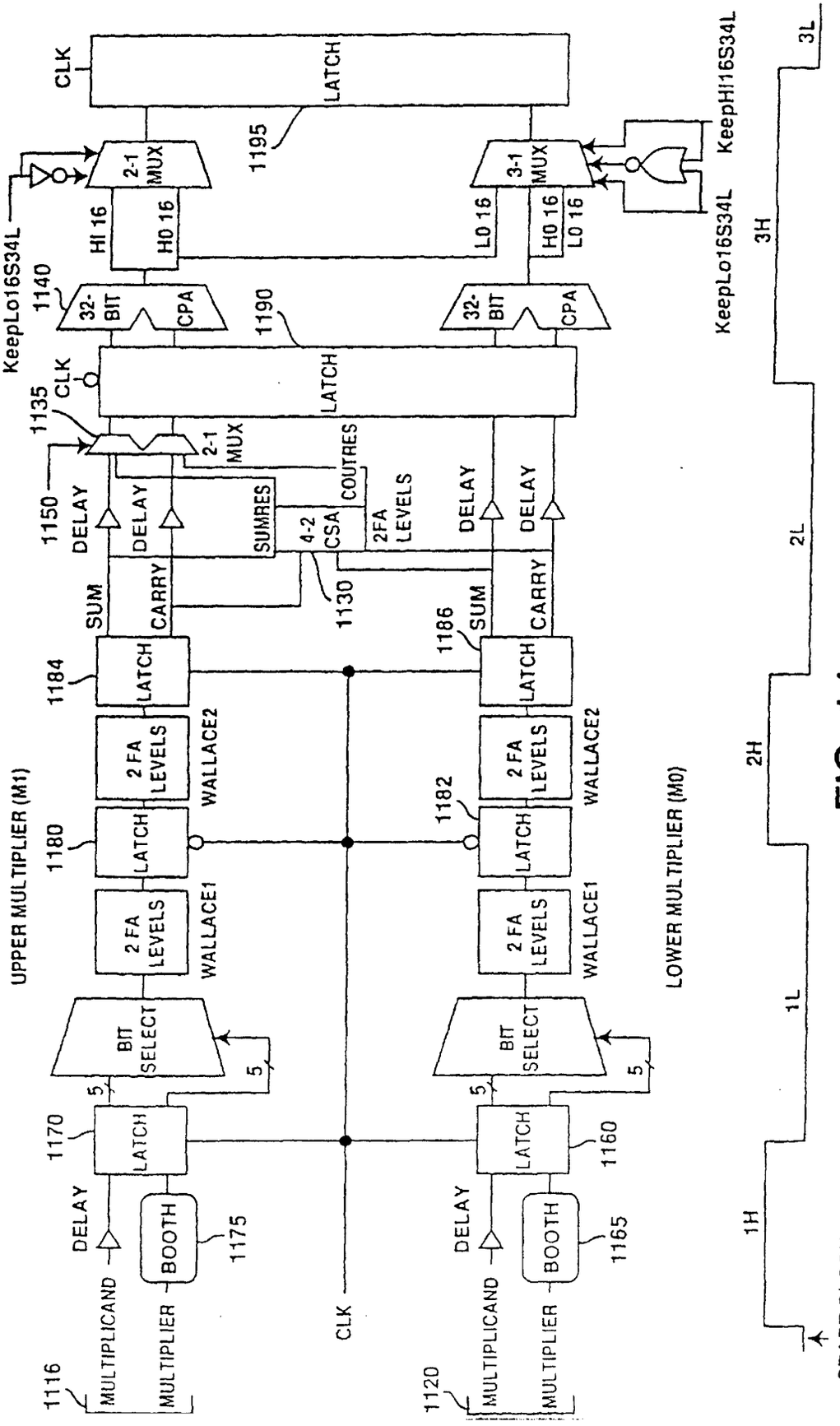

CLK
LATCH
1195
2-1 MUX
3-1 MUX
HI 16
HO 16
LO 16
KeepLo16S34L
KeepHi16S34L
1140
32-BIT CPA
1190
1135
1150
DELAY
SUMRES
4-2 CSA
COUTRES
2FA LEVELS
1130
SUM
CARRY
1184
1186
2 FA LEVELS
WALLACE2
WALLACE1
1180
1182
BIT SELECT
1170
1160
BOOTH
1175
1165
MULTIPLICAND
MULTIPLIER
1116
1120
UPPER MULTIPLIER (M1)
LOWER MULTIPLIER (M0)
3H
3L
2H
2L
1H
1L
START CLOCK

FIG. 11